

# **СПРАВОЧНИК**

## **ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ**

**В ТРЕХ КНИГАХ**

**1**

**СИСТЕМЫ ОБЩЕНИЯ  
И ЭКСПЕРТНЫЕ СИСТЕМЫ**

**2**

**МОДЕЛИ И МЕТОДЫ**

**3**

**ПРОГРАММНЫЕ  
И АППАРАТНЫЕ СРЕДСТВА**

# **СПРАВОЧНИК**

## **ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ**

**КНИГА**

**3**

### **ПРОГРАММНЫЕ И АППАРАТНЫЕ СРЕДСТВА**

ПОД РЕДАКЦИЕЙ  
В. Н. ЗАХАРОВА  
В. Ф. ХОРОШЕВСКОГО

**Scan Pirat**



МОСКВА „РАДИО И СВЯЗЬ”  
1990

ББК 32.81  
И 86  
УДК 681.3→007

Рецензент: проф. В. А. Горбатов

**Редакция литературы по информатике и вычислительной технике**

**Искусственный интеллект: В 3-х кн. Кн. 3. Программные И86 и аппаратные средства: Справочник/ Под ред. В. Н. Захарова, В. Ф. Хорошевского. — М.: Радио и связь, 1990.—368 с.: ил.**

**ISBN 5-256-00366-6 (кн. 3).**

Приводится классификация программных средств для интеллектуальных систем. Описываются конкретные языки представления знаний и манипулирования ими, базовые языки, используемые в интеллектуальных системах, системы поддержки разработки интеллектуальных систем, ЭВМ с высоким уровнем интерпретации языков, спецпроцессоры баз данных, знаний и логического вывода для ЭВМ пятого и последующих поколений.

Для специалистов в области управления, информационных систем и вычислительной техники, использующих методы искусственного интеллекта.

И 1402070000-146 29-30  
046(01)-90

**ББК 32.81**

**ISBN 5-256-00366-6 (Кн. 3)**  
**ISBN 5-256-00756-4**

© Коллектив авторов, 1990

## Предисловие

В настоящей книге излагаются вопросы использования существующих и перспективных инструментальных средств проектирования систем, ориентированных на работу со знаниями. Под инструментальными средствами понимается весь комплекс программно-аппаратных средств, позволяющих разработчику строить и совершенствовать создаваемые им специализированные интеллектуальные системы. Другими словами, это совокупность методов и средств, использование которых позволяет осуществить переход от постановки некоторой задачи к реализации системы, которая задачу решает.

Накопленный за прошедший период во многих странах опыт решения различных интеллектуальных задач показал, что современные универсальные ЭВМ, несмотря на постоянное совершенствование, недостаточно эффективны. Проблема построения специализированных вычислительных систем и аппаратных средств для решения таких задач совершенно очевидна и привлекает внимание исследователей многих стран.

Разработка программного обеспечения для систем, ориентированных на работу со знаниями, так же, как и разработка моделей и систем представления знаний, часто инициирует появление новых языковых средств. Так, для информационной системы GUS был разработан специальный язык представления знаний KRL (Knowledge Representation Language), а интеллектуальная система NUDGE вызвала к жизни библиотечную систему над Лиспом FRL (Frame Representation Language). Другим подобным примером являются язык представления лингвистических знаний и язык спецификации проблем типа Утопист. Эти и другие языки представления знаний нашли отражение в данной книге.

Наряду с развитием языковых средств в последние годы велись активные исследования в области создания систем программного обеспечения поддержки разработок, включающих подсистемы данных проекта, управляющие программы, подсистемы автоматизации программирования, подсистемы отладки программных комплексов и др. В этой области пока получены лишь предварительные результаты (некоторые из них изложены в настоящей книге).

Успехи современной микроэлектронной технологии создают предпосылки для построения относительно дешевых устройств, выполняющих весьма сложные функции, реализуемые аппаратным, а не программным путем. Возникает задача построения высокопроизводительных специализированных вычислителей с максимально развитым аппаратным обеспечением алгоритмов символьной обработки информации. В связи с этим здесь представлены некоторые материалы, на первый взгляд не имеющие непосредственного отношения к разработке интеллектуальных систем: специализированные вычислительные структуры, ассоциативные параллельные процессоры, однородные структуры, аperiodическая схемотехника. В них отражены результаты исследований последних лет в области эффективных архитектур вычислительных систем и микроэлектронной схемотехники, перспективные с точки зрения их использования в разработках аппаратных средств поддержки интеллектуальных систем.

При отборе материала редакторы старались учесть заинтересованность различных категорий читателей, как специалистов, работающих в области создания

сложных интеллектуальных систем и интересующихся инструментальными средствами поддержки их разработок, так и тех, которые хотят познакомиться с этой инженерной областью знаний.

Отдельные разделы книги написаны разными авторами, придерживающимися в ряде случаев различных точек зрения на преимущество того или иного метода или подхода. Отсутствие установившейся общепринятой терминологии в области разработки систем, ориентированных на знания, привело к тому, что авторы отдельных разделов иногда по-разному понимают одни и те же термины. При редактировании материалов ряд неоднозначностей был устранен, за исключением тех случаев, когда, по мнению редакторов, у авторов имелись достаточно веские причины для отстаивания своих точек зрения на рассматриваемые проблемы.

Большую помощь на всех этапах подготовки рукописи к печати оказали сотрудники отдела Проблем искусственного интеллекта Вычислительного центра АН СССР О. А. Белова, В. Н. Дембовская и С. М. Ефимова.

*В. Н. Захаров  
В. Ф. Хорошевский*

## Глава 1.

# Базовые средства программирования для интеллектуальных систем

## 1.1. Средства поддержки разработки интеллектуальных систем

*А. Г. Красовский, В. Ф. Хорошевский*

В области разработки интеллектуальных систем наиболее проработанным этапом является реализация программных проектов. Такая ситуация объясняется тем, что по идеям и методам этот этап близок к автоматизации программирования — одной из основных проблем использования средств вычислительной техники. Здесь уже в течение многих лет применяется обширный арсенал языков программирования высокого уровня, ориентированных на удобную и эффективную реализацию различных классов задач, а также широкий спектр трансляторов, обеспечивающих получение качественных исполнительных программ. Все шире используются на современном этапе и методы автоматического синтеза программ. Обычным становится применение языково-ориентированных редакторов и специализированных баз данных, в рамках технологии программирования уже практически сформировалась концепция окружения разработки сложных программных продуктов. Именно это окружение и определяет те инструментальные средства, которые доступны разработчикам.

Как и в случае классического программирования, в общей проблеме окружения, ориентированного на создание интеллектуальных систем, надо различать аппаратную и программную обстановку. Доступная разработчику аппаратура является тем базисом, на котором создаются, с одной стороны, инструментальные средства, а с другой — и сами системы, ради которых эти инструментальные средства разрабатываются. Поэтому аппаратная обстановка в значительной мере определяет архитектуру и функциональные возможности будущих систем, а также инструментальные средства для их разработки.

В настоящее время большая часть интеллектуальных систем реализуется на персональных ЭВМ (ПЭВМ) и специализированных рабочих станциях. Вместе с тем активное внедрение интеллектуальных систем в практику требует переноса их и на традиционные ЭВМ. Поэтому в рамках архитектуры инструментальных средств самое пристальное внимание уже в ближайшем будущем будет уделяться резидентным и кросссистемам — архитектурам, уже освоенным в рамках классической технологии программирования [Липаев и др., 1983].

*Резидентные системы* поддержки обеспечивают создание программных комплексов для тех же вычислительных средств, на которых они сами реализованы. Преимущества резидентных систем определяются прежде всего удобством использования единой программно-аппаратной среды для всех этапов разработки и сопровождения. Часто в резидентных системах средства поддержки разработки реализуются на базе типовых (штатных) средств программного обеспечения

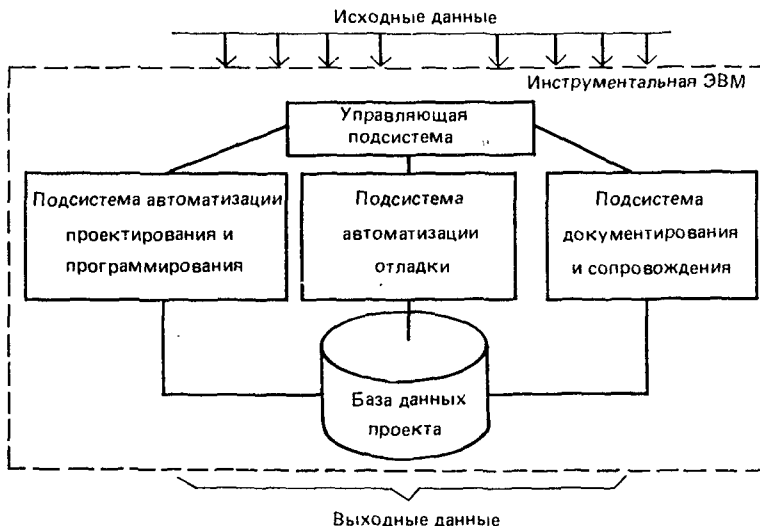


Рис. 1.1. Общая схема типовой технологической системы поддержки разработки

и/или путем прямого включения в соответствующую инструментальную среду.

*Кроссистемы* позволяют проектировать программные комплексы для ЭВМ, не совместимых с инструментальными средствами. В результате на целевой (рабочей) ЭВМ специальные технологические программы могут вообще отсутствовать либо включать минимальный набор средств, поддерживающих собственно развертывание системы. Эффективность кроссистем и перспективы их использования в связи с появлением заказных СБИС значительно повышаются. По мере становления индустрии баз знаний можно ожидать, что кроссистемы станут основным средством для тиражирования знаний в конкретные прикладные системы.

Технологическая система поддержки разработки должна поддерживать все основные этапы создания проектируемого программного комплекса. Для достижения этой цели в общей структуре типовой технологической системы (рис. 1.1) обычно выделяют базу данных проекта, подсистемы автоматизации проектирования и программирования, подсистемы автоматизации отладки, документирования и сопровождения, а также управляющую подсистему. В полном объеме базовая система поддержки разработки, как правило, воплощается в программных комплексах на достаточно мощных инструментальных ЭВМ и содержит сотни тысяч команд объектного кода.

*База данных проекта* предназначена для упорядоченного хранения и корректировки информации, отражающей текущее состояние и изменения в проектируемой программной системе.

*Подсистему автоматизации проектирования и программирования* составляют языки спецификации программного продукта и языки программирования, обеспечивающие соответствующими трансляторами и средствами подготовки исходных текстов. Для оттранслированных подсистем и модулей формируются соответствующие паспортные данные, а результаты трансляции помещаются в отдельную базу данных. Окончательная сборка разрабатываемой системы осуществляется редакторами связей целевой или (в случае кроссистем) инструментальной ЭВМ.

*Подсистемы автоматизации отладки, документирования и сопровождения* в современных системах поддержки разработок составляют существенную долю общего объема и включают (для подсистемы отладки, например) модули планирования отладки и модули накопления статистики о прохождении тестовых про-

грамм, что необходимо для оценки производительности разрабатываемой системы и ее сопровождения в процессе эксплуатации.

*Управляющая подсистема* предназначена для обеспечения всех режимов функционирования технологической системы, а также для организации накопления, хранения и обработки информации в базе данных. В функции управляющей подсистемы (в случае кросссистемы) входит и начальная настройка системы поддержки на тип целевой ЭВМ. В развитых технологических системах в состав управляющей подсистемы входит также модуль контроля разработки, осуществляющий сбор, обобщение и выдачу информации о ходе разработки и результатах работы каждого исполнителя.

### Специфика инструментальных сред интеллектуальных систем

Спецификой интеллектуальных систем является их ориентация на эксплицитное представление знаний о предметной области, в которой они будут работать. Такой подход на первый план в инструментальных средах поддержки разработки интеллектуальных систем выдвигает проблему создания адекватных средств представления знаний. Данный подход предполагает явно или неявно использование следующей парадигмы создания инструментальных средств. Сначала выбирается и/или создается язык представления знаний и соответствующий транслятор с этого языка; затем вокруг выбранного языка формируется программное окружение, поддерживающее процессы подготовки и трансляции программ на этом языке. Таким образом, в первую очередь создается подсистема автоматизации проектирования и программирования интеллектуальных систем.

На современном этапе выбор языка означает и выбор системы программирования, с которой в инструментальную среду имплементируются и средства отладки, и средства ведения архивов, и (неявно) средства сопровождения разрабатываемых систем. Язык программирования строго диктует не только программные, но и проектные, а в дальнейшем и технологические решения. Таким образом, подсистема управления ходом разработки, которая является одной из главнейших и в конечном счете определяет успех всего проекта, по существу, выносится из инструментальной системы на уровень организационных мероприятий. Разработчик прикладной интеллектуальной системы в качестве компенсации получает эффект быстрой прототипизации. Развитая система программирования позволяет, по крайней мере на первых порах, не заботиться специально об основных технологических аспектах ведения разработки. Однако при этом знания о том, как же собственно осуществляется разработка прикладной системы, становятся *know how* самого разработчика, а создание системы искусством. На начальных стадиях разработки интеллектуальных систем и/или в небольших проектах использование развитых сред автоматизации программирования является самым быстрым способом получения демонстрационного прототипа.

Следующим этапом в развитии инструментальных средств является ориентация на явное включение в технологические среды знаний. При этом все знания можно разделить на знания о процессах программирования интеллектуальных систем и технологические. Развитие инструментальных сред первого направления связано с созданием интеллектуальных систем автоматизированного, а в конечном счете и автоматического синтеза исполнительных программ, а второго — с реализацией интеллектуальных систем поддержки разработок.

В настоящее время более активно ведутся работы по первому направлению. И это естественно, так как современный инструментарий интеллектуальных систем является, по существу, эволюционным развитием систем автоматизации программирования. При этом основная доля мощности и интеллектуальности такого инструментария связывается не с его архитектурой, а с функциональными возможностями отдельных компонентов той или иной технологической среды. Большое значение при разработке инструментария уделяется и удобству сопряжения отдельных компонентов. Пожалуй, именно здесь получены впечатляющие результаты и именно здесь наиболее широко используются последние достижения теории и практики программирования. Вместе с тем подавляющее большинство



современных инструментальных систем (instrumental tool box, knowledge engineering environment, shell и т. п.) не «знают», что проектирует и реализует с их помощью пользователь. И с этой точки зрения можно сказать, что все такие системы являются не более чем «суидучками» с инструментами, успех использования которых определяется искусством работающего с ними мастера.

Такой инструментарий оправдывает себя при создании экспериментальных интеллектуальных систем, но неприемлем на уровне массового внедрения методов и средств инженерии знаний в практику и тиражирования баз знаний. Инструментальные системы нового поколения будут опираться на знания о технологии проектирования, реализации и сопровождения интеллектуальных систем в той же мере, в какой современный инструментарий базируется на знаниях о процессах трансляции.

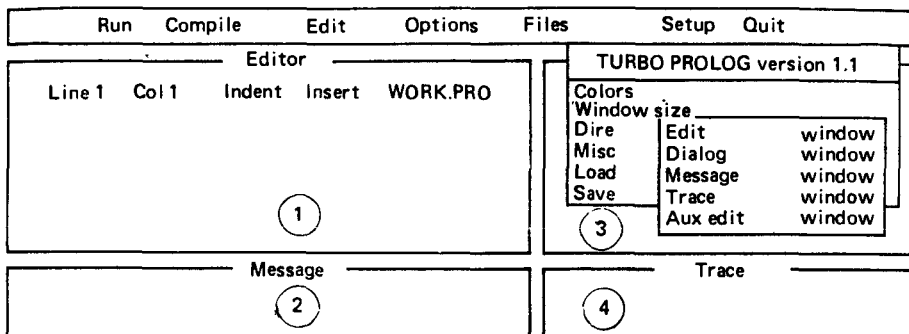
Выше мы ввели в рассмотрение три уровня технологических средств для создания интеллектуальных систем: подсистемы автоматизации программирования; инструментальные «суидучки» и интеллектуальные инструментальные системы. Остановимся на конкретных примерах указанных типов сред.

### Инструментарий интеллектуального программирования

Наиболее яркими представителями интегрированных сред автоматизации программирования являются TURBO-среды фирмы Borland — пионера и лидера в этой области [Borland, 1985, 1986, 1987]. Для всех TURBO-сред характерны очень быстрая компиляция исходных текстов, достаточно развитые встроенные экранные редакторы входных программ, удобные интерактивные средства отладки и развитые библиотеки периода исполнения (run-time libraries). И все эти компоненты связаны в единую сбалансированную среду. Наиболее мощной в TURBO-семействе является среда Turbo C [Borland, 1987], в рамках которой реализованы практически все основные идеи интерактивного программного окружения на основе языка высокого уровня и, по-видимому, впервые обеспечена реальная возможность стыковки Turbo-программ на Паскале, Прологе и Си. Пользователям предоставлены, кроме того, возможности удобного имплантирования программ, созданных вне Turbo C, в сложные программные комплексы, разрабатываемые на его основе. И, наконец, фирмой Borland выпущены на рынок мощные пакеты (TURBO TOOLBOX), поддерживающие основные технологические потребности пользователей.

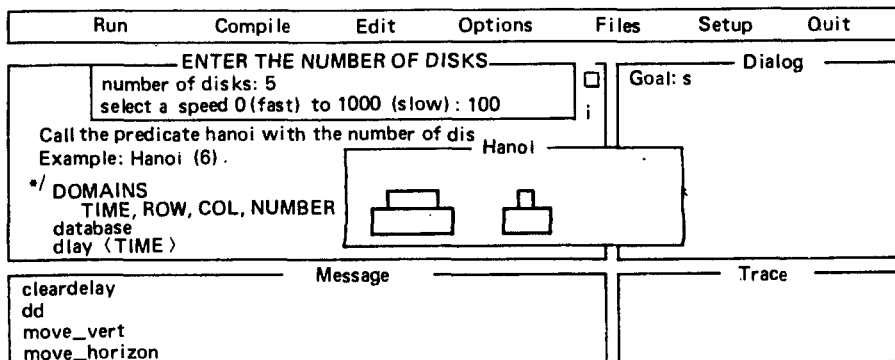
Покажем возможности TURBO-среды на примере языка Turbo Prolog [Borland, 1986]. В целом система Turbo Prolog компактна и дружелюбна к пользователю. Весь интерфейс базируется на использовании многооконного режима и простейших иерархических меню. В системе поддерживаются пять базовых типов работ (рис. 1.2,а): запуск Пролог-программ на выполнение (Run), их компиляция (Compile) и редактирование (Edit), взаимодействие с файловой и операционной системами ПЭВМ (Files), а также окончание работы с TURBO-средой (Quit). Кроме того, в основном меню представлены две дополнительные альтернативы — варианты (Options) и установка параметров (Setup). Первая связана, например, с уточнением режима трансляции Пролог-программ, вторая позволяет изменять параметры окружения, в первую очередь, конечно, изменяя расположение на экране стандартных окон системы. Всего таких окон (включая основное меню) пять: окно редактора, окно диалога, окно вывода сообщений и окно трассировки Пролог-программ. Дополнительно к этим основным окнам пользователь может определять свои окна, располагая их на экране наиболее удобным для себя образом. Пример использования окон в Пролог-программе показан на рис. 1.2,б. Здесь пользователь в программу решения задачи о Ханойских башнях ввел два дополнительных окна: окно приема начальных условий и окно визуализации процесса перемещения дисков.

Краткое рассмотрение возможностей среды Turbo Prolog показывает, что вся ее «интеллектуальность» заключена во входном языке. Turbo Prolog удобен для компиляции, и в нем при необходимости можно достаточно эффективно реализо-



Use first letter of option or select with → or <—

a)



Use first letter of option or select with → or <—

b)

Рис. 1.2. Использование иерархических меню (а) и многооконного режима (б) при выполнении программы Пролог:

1 — окно редактора Пролог-программы; 2 — окно вывода сообщений; 3 — окно диалога с пользователем; 4 — окно трассировки Пролог-программы

вать не только логический вывод, но и остальные компоненты разрабатываемых систем.

**Система EXSYS.** Включение специализированных языков естественно влечет за собой и определенные изменения в технологической поддержке их использования. Эти изменения, в первую очередь, касаются разработки специальных языково-ориентированных редакторов, которые позволяют пользователю оперировать в процессе реализации не столько категориями языка программирования, сколько понятиями предметной области, для работы в которой создается прикладная система. Примерами подобных сред служат подавляющее большинство инструментальных «сундучков» и систем-оболочек для создания экспертных систем (ЭС). Например, система EXSYS [EXSYS, 1985] ориентирована на новичков в области создания ЭС. Может быть, именно поэтому основные усилия ее разработчиков были направлены на то, чтобы сделать систему, которая бы активно помогала пользователю. Этой цели прежде всего служит интерактивное руководство по системе EXSYS, построенное в виде уроков-демонстраций, а также система поясняющих

текстов, привязанных к определенным точкам процесса проектирования ЭС. Подсистема обучения пользователей базируется на применении интерактивного синтаксически-ориентированного редактора правил, который и является ядром системы EXSYS. Кроме собственно редактора в это ядро входит и интерпретатор правил, поддерживающий одну из трех стандартных стратегий вывода решений. Его настройка на определенную стратегию осуществляется путем опроса пользователя в режиме меню при входе в систему. Имеется в EXSYS и компилятор правил, используемый для повышения быстродействия работы прикладной продукционной системы. Возможна трассировка правил, поддерживающая простейшие объяснения типа «почему?» и «как?».

Язык представления знаний системы EXSYS опирается на конструкцию типа IF-THEN-ELSE. Синтаксическая структура правил следующая:

RULE#номер-правила	
IF	условие
THEN	выражение
[ELSE	выражение]
[NOTE	N-текст]
[REFERENCE	R-текст]

где условие — логическое выражение, в котором могут использоваться связи И, ИЛИ, НЕ (при этом И-связка используется по умолчанию); выражение — последовательность операторов и обращений к внешним процедурам; N-текст — пояснение правила, используемое в подсистеме объяснения EXSYS; R-текст — ссылка на источник информации для правила.

Обязательными элементами правила являются RULE, IF THEN, остальные элементы факультативны. Вместе с тем, если пользователь хочет, чтобы подсистема объяснений при обращении к ней выдавала хоть какую-то информацию, он должен использовать NOTE-часть.

Процесс проектирования правил с помощью синтаксически-ориентированного редактора системы EXSYS показан на рис. 1.3. По существу, данный редактор пользуется тремя основными окнами-полями: меню-подсказкой, где перечислены все доступные пользователю в данный момент команды и выведена подсказка, чего ждет от него сейчас система (1); полем определения квалификаторов, которые, в конечном счете, являются переменными проектируемого продукционного правила (2), и полем формирования (редактирования) правила, на котором система и собирает в соответствии с приведенной выше синтаксической схемой очередное правило (3). Кроме того, EXSYS-редактор имеет стандартные возможности просмотра, редактирования и удаления правил. Как и всегда в случае таких простых средств представления знаний, более искушенным пользователям системы EXSYS предлагается подключать внешние подпрограммы, в том числе

RULE NUMBER: 3

IF:

The price of the printer is a major consideration

THEN:

Model A — Probability = 9/10

Model B — Probability = 5/10

Model C — Probability = 3/10

Qualifier #3

The price of the printer is

- 1 a major consideration
- 2 of concern, but of less importance then functionality
- 3 of little concern

Enter a qualifier number, New qualifier (N), Find text (F), Last qualifier (L), Repeat cond (R), Choice (C), Math/Variable (M), Help (H) or (ENTER) when done

Рис. 1.3. Проектирование продукционных правил в среде системы EXSYS

и для обмена информацией с другими системами. Однако самой системой под-держивается обмен через последовательные файлы ДОС.

Таким образом, система EXSYS точно соответствует классу простых и недо-рогих систем для новичков. С ее помощью в общих чертах можно познаться с некоторыми аспектами проектирования ЭС и даже сделать самому простейший демонстрационный прототип, но на базе данной системы нельзя до конца понять ни что такое технология создания ЭС, ни спроектировать сложную базу знаний.

Система GURU. Существенно более широкими возможностями, чем EXSYS, обладает интегрированная оболочка GURU — продукт фирмы Micro Data Base Systems, Inc. [MDBS, 1986]. По существу система GURU состоит из четырех достаточно независимых компонентов, объединенных общей программной обста-новкой и единым стилем взаимодействия с подсистемами. В качестве единой парадигмы интерфейса здесь применяются система иерархических меню и мно-гоокопный режим работы. На самом верхнем уровне пользователю предлагаются следующие опции: экспертные системы, естественный язык, управление данными, изменение окружения и выход. Выбрав одну из них, он попадает в соответствующий блок или покидает систему.

Функциональные возможности системы GURU в целом достаточно традицион-ны для современных интегрированных сред [Брибри, 1988]. Это и работа с элек-тронными таблицами, и обработка данных с использованием системы управления БД (СУБД), в качестве которой здесь выступает Knowledgepan, и несложный калькулятор. Как и в системе EXSYS, проектирование ЭС с помощью системы GURU в конечном счете опирается на интерактивный редактор и компилятор правил. Однако выход в набор правил требует здесь использования существенно более длинной цепочки меню. Редактор правил в системе GURU многоокопный (рис. 1.4).

В языке представления знаний GURU могут поддерживаться различные стратегии разрешения конфликтного множества в процессе вывода, используются правила типа IF-THEN, с каждым правилом может быть связан свой собствен-ный поясняющий текст. В целом управление редактором несложно и потому до-ступно даже для пользователей невысокой квалификации. Вместе с тем неявно предполагается, что пользователь хорошо знает язык представления знаний GURU и потому несет полную ответственность за корректное использование тех или иных его конструкций. Кроме выхода в редактор правил из того же меню пользователь может попасть и в блок описания/редактирования переменных, и в блоки определения целевых атрибутов и начальных значений переменных, и в блок печати проектируемой базы знаний. Выход из меню этого уровня автома-тически инициирует еще одно подменю, опции которого связаны с сохранением и компиляцией базы знаний.

Несмотря на то, что система GURU достаточно мощная, ее нельзя исполь-зовать в качестве адекватного инструмента для построения широкого класса ЭС. Это связано, во-первых, с большим числом допустимых на каждом уровне опций, в которых быстро «тонут» начинающие, и, во-вторых, с жесткой регламентацией использования «анфилада» из меню для попадания в нужный блок, что раздражает искушенных. Вместе с тем не видно, что получает пользователь GURU в каче-стве компенсации. Если это более или менее удобный редактор правил, то этого мало. Если же это все достоинства работы в рамках интегрированной среды обработки данных, то, может быть, правильнее было бы исключить из системы саму опцию «Экспертные системы» и вернуться к тому классу, под который и разрабатывалась GURU сначала.

Системы EXSYS, GURU [Harmon, 1987] — коммерческие продукты второй волны в инструментальном буме программного обеспечения интеллектуальных систем. Системы ART [ART, 1984], KEE [Florenti, 1987] и Knowledge Craft [CARNEGIE, 1987] — это безусловно интегрированные среды поддержки разра-ботки интеллектуальных систем и в первую очередь ЭС. И вместе с тем для этих систем характерно не эклектичное объединение различных полезных блоков, а тщательно сбалансированный их отбор, что позволяет сделать первые шаги от системы автоматизации программирования к технологическим системам поддерж-

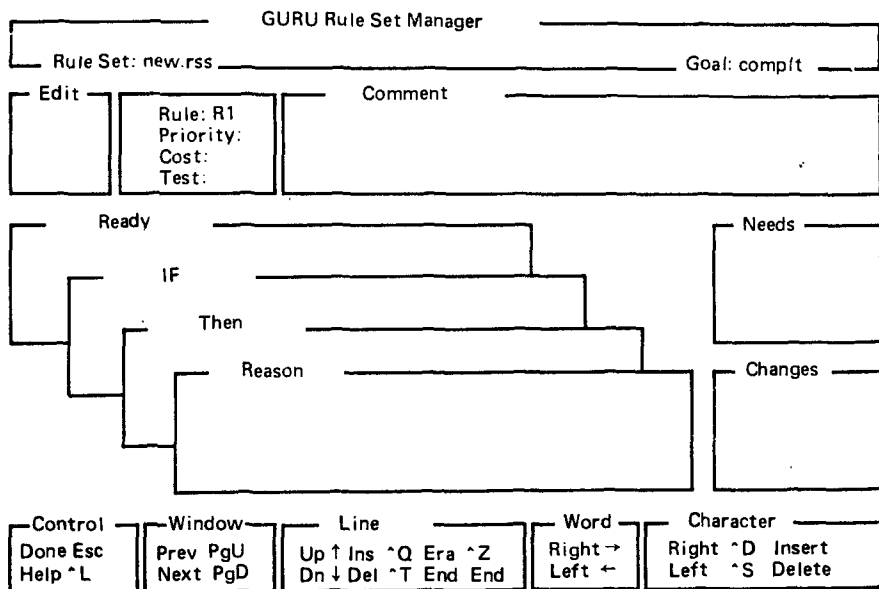


Рис. 1.4. Архитектура редактора правил системы GURU

ки разработки сейчас экспертных, а в ближайшем будущем и других интеллектуальных систем.

**Система KEE (Knowledge Engineering Environment).** Разработана одной из первых в этой области фирм — фирмой IntelliCorp. Inc., которую возглавляют такие известные ученые, как Э. Фейгенбаум и Р. Файкс. В создании системы активное участие принимал и И. де Клир из Исследовательского центра «Ксерокс Поло Алто». Первая версия системы была анонсирована в 1983 г., версия 3.0 — летом 1986 г. Для системы KEE в полном объеме требуются рабочие станции типа Symbolics 3600, TI Explorer, Xerox 1100 или Sun-3, достаточно развитые версии KEE функционируют и на ПЭВМ типа IBM PC/AT. Примерно в то же время, что и IntelliCorp., была образована фирма Automated Reasoning Tool Inc., выпускающая инструментальные оболочки ART. Что же касается системы Knowledge Craft, то это продукт недавно образованной, но уже зарекомендовавшей себя с лучшей стороны фирмы Carnegie Group Inc. Системы ART и Knowledge Craft предъявляют примерно те же требования к аппаратуре, что и KEE. Сравнимы все три эти системы и по функциональным возможностям, а различаются они средствами представления знаний и стратегиями вывода решений.

Система KEE представляет собой набор средств представления и манипулирования знаниями, в ней имеется мощный графический интерфейс на базе пиктограмм. В общем случае можно считать, что система KEE состоит из двух компонентов: фрейм-ориентированной подсистемы моделирования и подсистемы вывода решений на основе правил. Факты представляются фреймами, с фактами могут работать продукционные правила. Вывод решений осуществляется с помощью прямого и/или обратного вывода. На таком уровне KEE мало отличается от других систем, мощных и слабых, хороших и плохих. При более детальном анализе системы KEE становится понятным, почему несмотря на высокую цену она активно используется для реализации прикладных интеллектуальных систем.

В системе KEE слоты фреймов содержат не только данные и присоединенные процедуры, но и описания графических растров (bit-maps), необходимые для

единообразной работы с графическими примитивами, поддерживается наследование свойств. В системе возможна динамическая генерация фреймов-экземпляров по фреймам-прототипам с помощью посылки сообщений в объектно-ориентированном стиле. Для поддержки такого режима в системе KEE введено два типа слотов: собственные (own slots) и коллективные (member slots). Кроме того, слоты фреймов могут содержать ограничения на их заполнение. В качестве таких ограничений может быть, в частности, декларация «неизвестно» (Unkown), которая учитывается специальным образом при работе механизмов дедуктивного вывода. Формирование фактов поддерживается мощным графическим редактором, где с помощью пиктограмм можно легко и быстро определить любые фрейм-структуры с наследованием типа ISA. Если же разработчику требуется другое наследование, он должен поддержать его собственными Лисп-программами.

Правила в системе KEE имеют обычную синтаксическую структуру типа IF THEN. Их естественность с точки зрения пользователя достигается тщательно продуманным набором предикатов и действий. В правилах активно используются факты из предметной области и образцы. В процессе формирования и разрешения конфликтного множества переменные образцов получают значения, которые в дальнейшем могут использоваться для явного или неявного управления выводом. Сами стратегии вывода могут указываться явно с помощью специальных утверждений (например, ASSERT для прямого вывода и QUERY для обратного) либо программироваться при создании прикладной системы самим разработчиком. Вместе с тем пользователям предоставляется мощная встроенная стратегия ATMS (Assumption-based Truth Maintenance Strategy) [Kleer, 1986].

Что касается формирования и отладки баз знаний, то в системе KEE принят обычный подход систем автоматизации программирования. В ней пользователю предоставляется мощный графический редактор правил, используемый для начального ввода продукции и коррекции их в процессе отладки, и средства графической трассировки вывода решений, которые позволяют инженеру знаний легко ориентироваться во взаимодействии сотен и тысяч правил. Сами же эти компоненты лишь дружелюбны к пользователю, но не интеллектуальны и практически ничего не «знают» о тех процессах, которые они поддерживают. Такая же ситуация и в системах ART и Knowledge Craft.

В настоящее время уже достаточно четко просматриваются два направления: применение методов искусственного интеллекта для разработки программного обеспечения и переход к системам поддержки разработки прикладных баз знаний, базирующихся, в свою очередь, на метазнаниях о протекающих здесь процессах. Так как само программирование является одной из задач, требующих высокой степени интеллектуального участия человека, совершенно естественно, что было уже немало попыток применения методов искусственного интеллекта к процессу разработки программного обеспечения. Можно выделить три различных подхода к этой проблеме:

- трансформационный подход к генерации программ;
- моделирование процесса программирования;
- применение интеллектуальных систем для улучшения инструментария программирования.

Трансформационный подход имеет целью генерацию программного обеспечения непосредственно из спецификаций, при этом язык спецификаций и является языком представления знаний в области проектирования программ, как, например, в системе KBEmacs [Waters, 1985], обеспечивающей клише и планы для построения программ на языках Лисп и Ада с помощью всего нескольких команд. Клише представляет собой настраиваемую параметризованную процедуру для решений некоторой задачи, такой, например, как поиск. Пользователь может, указав параметры, вызвать клише и получить сразу целую часть программы. Хороший набор клише позволяет иметь словарь наиболее употребительных понятий среднего и даже высокого уровня. План обеспечивает пользователя удобными структурами для представления всей программы с целью выполнения над ней различных действий с помощью разнообразных инструментальных средств.

Реализация систем автоматического синтеза программ опирается на знания.

Знания о программе можно разделить на знания по технологии программирования и знания о конкретной проблемной области. Первый вид знаний связан с опытом в разработке программного обеспечения. В ряде исследований предпринимались попытки создания теории знаний по технологии разработки программного обеспечения с целью определения, каким образом эксперты-программисты понимают, проектируют, реализуют, тестируют, отлаживают, модифицируют и документируют программы [Argando et al., 1985]. Результаты этих исследований уже сейчас могут быть использованы для автоматизации некоторых сторон процесса разработки программного обеспечения. Второй вид знаний специфичен для каждой проблемной области. Эти знания могут использоваться для автоматизации проверки соответствия различных программных решений физическим законам данной проблемной области, а также для проверки и тестирования проектных решений.

Различные методы искусственного интеллекта могут значительно увеличить мощность существующих инструментальных средств, например, в области разработки, оценки и проверки требований. Очень часто пользователь не знает тех возможностей и особенностей, которые он хотел бы иметь от программной системы. При этом процесс разработки требований напоминает процесс приобретения знаний. Таким образом, средства накопления знаний могут использоваться при формировании пользователем своих требований.

Хотя модели жизненного цикла для большинства экспериментальных систем искусственного интеллекта существенно различаются, проблемы здесь возникают в общем-то одни и те же. Ряд программных систем поддерживает весь жизненный цикл. Таковы, например, системы Gandalf [Habermann et al., 1986], Genesis [Ramamoorthy et al., 1985], Saga [Kerlis et al., 1985], которые представляют собой экспериментальные попытки создания систем, базирующихся на знаниях, со стороны классических технологических сред поддержки разработки. Вместе с тем говорить сегодня о выходе таких систем в практику еще рано. Сейчас опережающими темпами развиваются интеллектуальные инструментальные средства приобретения знаний и сопровождения баз знаний. «Узким местом» современных интеллектуальных систем является формирование, отладка и сопровождение баз знаний. Процессы эти плохо изучены и формализованы и крайне слабо поддерживаются инструментальными средствами [Anjewierden, 1987]. Из пяти уровней в приобретении знаний — лингвистического, концептуального, эпистемологического, логического и реализационного [Brachman, 1980] — лишь последние два в поле зрения основной массы разработчиков инструментария для интеллектуальных систем. Вместе с тем основные трудности ждут инженеров знаний именно на начальных этапах, где формируется так называемое поле знаний [Гаврилова, 1987]. Для поддержки первых трех уровней используются редакторы. Однако их архитектура и принципы проектирования существенно отличаются от «традиционных» синтаксически-ориентированных редакторов для языков представления знаний.

В качестве примера остановимся на редакторе протоколов PED (Protocol Editor) системы KADS (Knowledge Acquisition Development System), разрабатываемой в настоящее время в рамках проекта ЭСПРИТ 1998 в Амстердамском университете [Wielinga et al., 1986]. Это одна из немногих на сегодняшний день подсистем инструментальной поддержки уровня идентификаций знаний. Редактор PED опирается на парадигму гипертекста [Nelson, 1980], в рамках которой предполагается, что исходные текстовые протоколы структурируются путем выделения на экране необходимых по мнению инженера знаний фрагментов, аннотации их и определения связей между выделенными фрагментами. Кроме того, PED поддерживает объединение фрагментов в произвольные группы и возможность определения связей между такими группами. И, наконец, в данном редакторе имеются простые средства выбора и визуализации выделенных фрагментов и/или групп фрагментов с учетом определенных между ними связей. Результаты работы редактора протоколов являются в системе KADS исходными данными для редактора CE (Concept Editor), поддерживающего уровень концептуализации в процессе извлечения знаний. Основной задачей этого компонента системы приобретения знаний является дальнейшая структуризация и частичная формализа-

ция описаний протоколов, полученных в рамках PED. Редактор CE «знает», что такое концепт, что концепты могут иметь атрибуты, которые, в свою очередь, могут иметь значения. В базе знаний редактора CE описывается и то, что концепты могут быть связаны определенными отношениями, среди которых выделено в качестве системного отношения ISA. Работа инженера знаний в редакторе CE заканчивается, когда сформированы необходимые для решения прикладных задач структуры представления знаний и определены связи между ними. Кроме того, на уровне редактора CE поддерживается и формирование лексикона предметной области, и глоссария используемых в ней понятий.

Эпистемологический уровень приобретения знаний в системе KADS поддерживается редактором концептуальных моделей (Conceptual Model Editor) и библиотекером моделей интерпретации IML (Interpretation Model Librarian). По существу это уже этап получения спецификаций, которые на следующих двух уровнях (логическом и реализационном) будут преобразованы сначала в описание на соответствующем языке представления знаний, а затем транслированы в машинное представление базы знаний.

Такова технологическая цепочка в интеллектуальной инструментальной системе приобретения знаний KADS. Это одна из первых систем данного типа. Но уже в ней сделана попытка объединения достижений «классической» технологии программирования и методов искусственного интеллекта. В будущем эта тенденция, на наш взгляд, будет проявляться все более определенно. И одним из примеров такого подхода является японский проект ЭВМ пятого поколения, в котором анонсирована основанная на знаниях поддержка разработки как технологии систематизации [Громов, 1985]. В этом проекте предпринята попытка использования всех трех подходов к поддержке разработки с использованием методов искусственного интеллекта. В проекте поставлена задача сделать возможным синтез программ из требований пользователей с автоматическим выбором алгоритмов из соответствующего банка. Для решения этой задачи предполагается развитие теории спецификации и верификации программ автоматического синтеза. Другой целью проекта является разработка системы, выполняющей роль консультанта при проектировании программного обеспечения, а также различных инструментальных средств для сопровождения и улучшения программ.

## 1.2. Языковые средства программирования

*В. Ф. Хорошевский*

Необходимость использования средств автоматизации и программирования интеллектуальных систем была осознана разработчиками уже давно. Оценивая этот процесс с современных позиций, в области автоматизации разработки интеллектуальных систем можно указать две тенденции. Первая как бы повторила классический путь средств автоматизации программирования: автокоды — языки высокого уровня — языки сверхвысокого уровня — языки спецификаций. Эту тенденцию можно назвать восходящей стратегией разработки средств автоматизации программирования интеллектуальных систем. Вторая тенденция, нисходящая, связывалась с созданием специальных средств, изначально ориентированных на определенные классы задач и методов. Представляется важным хотя бы кратко рассмотреть эволюцию средств автоматизации для создания систем, ориентированных на знания.

На первом этапе развития искусственного интеллекта (в конце 50-х — начале 60-х годов) не существовало языков и систем, ориентированных специально на области знаний. Появившиеся к тому времени универсальные языки программирования казались адекватным инструментом для создания любых (в том числе и интеллектуальных) систем, поскольку в этих языках можно выделить декларативную (описания) и процедурную (последовательности операторов, реализующих заданные алгоритмы над заданными описаниями) компоненты. Казалось, что на этой базе могут быть интерпретированы любые модели и системы представле-



ния знаний. Однако сложность и трудоемкость таких интерпретаций оказались настолько велики, что прикладные системы для реализации были недоступны. Проведенные несколько позже исследования в области технологии программирования [Йодан, 1979, Глушков и др., 1975, Липаев и др., 1983, Миллис, 1970] показали, что производительность труда программиста (количество операторов программного продукта, которое он производит на заданном временном интервале) остается постоянной независимо от уровня инструментального языка, на котором он работает, а соотношение между длиной исходной и результирующей программ примерно 1 : 10. Таким образом, использование адекватного инструментального языка повышает производительность труда разработчика системы на порядок, и это при одноступенчатой трансляции. Однако при создании интеллектуальных систем в настоящее время исходная программа на инструментальном языке, как правило, обрабатывается иерархической системой трансляторов, так что реальное повышение производительности труда еще больше. Все это подтверждает важность использования адекватных инструментальных средств. В потоке публикаций до реализации систем, ориентированных на знания, одним из самых частых словосочетаний является язык Лисп. Поэтому рассмотрим кратко эволюцию языка Лисп [McCarthy, 1978].

Язык Лисп разработан в Стэнфорде под руководством Дж. Маккарти в начале 60-х годов [McCarthy et al., 1963]. По первоначальному замыслу он должен включать наряду со всеми возможностями Фортрана средства работы с матрицами, указателями и структурами из указателей и т. п. Предполагалось, что первые реализации будут интерпретирующими, но в дальнейшем будут созданы компиляторы, транслирующие Лисп-конструкции в машинный код. К счастью, для такого проекта не хватило средств. К тому же к моменту создания первых Лисп-интерпретаторов в практику работы на ЭВМ стал входить диалоговый режим, и режим интерпретации естественно вошел в общую структуру диалоговой работы. Примерно тогда же окончательно сформировались и принципы, положенные в основу языка Лисп: использование единого спискового представления для программ и данных; применение выражений для определения функций; скобочный синтаксис языка. Процесс разработки языка завершился созданием версии Лисп 1.5 [McCarthy et al., 1963, Лавров и др., 1978], которая на многие годы определила путь его развития и совершенствования.

Если исходить из базового набора примитивов (CAR, CDR, CONS, COND и т. д.), Лисп является языком низкого уровня. И с этой точки зрения его можно рассматривать как ассемблер, ориентированный на работу со списковыми структурами. Поэтому на протяжении всего существования языка было много попыток его усовершенствования за счет введения дополнительных базисных примитивов и/или управляющих структур. Однако все эти изменения, как правило, не прививались в качестве самостоятельных языков. И причин здесь несколько. С одной стороны, в большинстве случаев создатели новых языков оставались в «лисповской» парадигме, не предлагая нового взгляда на программирование. С другой — новые языки, как правило, не имели собственной программной среды, а «жили» в Лисп-среде и поэтому воспринимались как часть этого языка. В новых своих редакциях Лисп быстро «усваивал» все ценные изобретения конкурентов [McCarthy, 1978, Ангелова и др., 1984]. Наконец, немаловажную роль в распространении Лиспа и утверждении его в качестве основного языка интеллектуальных систем сыграли авторитет Стэнфордской школы и лично Дж. Маккарти в области искусственного интеллекта, а также введение Лиспа как обязательного для изучения студентами языка во всех учебных заведениях США, связанных с проблематикой искусственного интеллекта.

После создания в начале 70-х годов мощных Лисп-систем MacLisp и InterLisp [Moop, 1984, Urgi, 1976] попытки создания языков искусственного интеллекта, отличных от Лиспа, но на той же идеологической основе, по-видимому, сходят на нет. И дальнейшее развитие языка как идет, с одной стороны, по пути его стандартизации (таковы, например, Standart Lisp [Barr et al., 1982], Franz Lisp [Nordstroem, 1978] и Common Lisp [GOLDEN, 1985]), а с другой — в направлении создания концептуально новых языков для представления и манипулирования

знаниями, погруженных в Лисп-среду [Ангелова и др., 1984]. В настоящее время Лисп реализован на всех классах ЭВМ, начиная с ПЭВМ и кончая высокопроизводительными вычислительными системами [McCarthy, 1978, Tello, 1985, Steele, 1982].

Основными диалектами языка Лисп, используемыми при разработке и реализации систем, ориентированных на знания, являются Interlisp и MacLisp. И та, и другая система наряду с традиционными для Лиспа атомами и списками обеспечивают возможность работы со списками свойств, массивами, строковыми переменными, целыми и вещественными числами различной разрядности. В указанных системах имеются мощные макросредства и развитые управляющие структуры. Лисп-трансляторы, входящие в состав соответствующих систем программирования, порождают объектный код, сравнимый по эффективности с кодом, полученным с помощью языков типа ассемблеров. Вместе с тем эти системы имеют и некоторые важные отличия. Так, в Interlisp основное внимание было уделено удобству системы для пользователя. Главный принцип разработчиков этого диалекта: все, что может иметь место в системе, должно естественно выражаться в терминах ее входного языка. Поэтому в Interlisp программисту доступно все. Он может переопределять любые, в том числе и встроенные, функции; задавать и переопределять реакции на ошибки; работать непосредственно с уровня входного языка с внутренними структурами интерпретатора и т. д. При этом система поддерживает свою целостность и работоспособность. В отличие от этого разработчики MacLisp основные усилия сосредоточили на эффективности. Этому служат указания, уточняющие способы обработки аргументов функций, а также экранирование от вмешательства программиста внутренних механизмов системы. За счет этих мер скорость работы MacLisp в 1,5—2,5 раза выше, чем Interlisp.

Новый толчок развитию Лиспа дало создание Лисп-машин. Кроме высокого быстродействия (первая же Лисп-машина работала в несколько раз быстрее, чем MacLisp на ЭВМ DEC-20) и огромной виртуальной списковой памяти, достоинством Лисп-машин является и то, что для них это единственный язык программирования. На нем написаны все системные программы, начиная с операционной системы и кончая всевозможными препроцессорами, и программы пользователя. Такая однородность значительно упрощает как разработку самих системных компонентов, так и взаимодействие с ними. По сути дела, на Лисп-машине стирается грань между системным и прикладным программным обеспечением. В настоящее время Лисп-машины выпускаются рядом фирм США, Японии и Западной Европы. Имеются положительные примеры разработки таких машин и в социалистических странах.

Лисп (а вернее, его современные диалекты) — не единственный язык, используемый для задач искусственного интеллекта. Уже в середине 60-х годов, т. е. на этапе становления Лиспа, разрабатывались языки, предлагающие другие концептуальные основы. Наиболее важными из них в области обработки символической информации являются Снобол [Griwold, 1978], разработанный в лаборатории Белла, и Рефал [Турчин, 1968], созданный в ИГМ АН СССР.

Один из них (Снобол) — язык обработки строк, в рамках которого впервые появилась и была реализована в достаточно полной мере концепция поиска по образцу (pattern matching). Язык Снобол был одной из первых практических реализаций развитой продукционной системы. Наиболее известная и интересная версия этого языка — Снобол-IV [Грисуолд и др., 1980]. Здесь техника задания образцов и работа с ними существенно опередили потребности практики. Может быть, именно это, а также политика активного внедрения Лиспа помешали широкому использованию языка Снобол в области искусственного интеллекта. По существу, он так и остался «фирменным» языком программирования, хотя концепции Снобола, безусловно, оказали влияние и на Лисп, и на другие языки программирования задач искусственного интеллекта.

В основу языка Рефал положено понятие рекурсивной функции, определенной на множестве произвольных символических выражений. Базовой структурой данных этого языка являются списки, но не односвязные, как в Лиспе, а двусторонние. Обработка символов ближе к продукционной парадигме. При этом

активно используется концепция поиска по образцу, характерная для Сиоболла. Таким образом, Рефал вообрал в себя лучшие черты наиболее интересных языков обработки символической информации 60-х годов. В настоящее время язык Рефал-2 используется для автоматизации построения трансляторов [Бычков и др., 1982; Грох и др., 1983], систем аналитических преобразований [Турчин и др., 1969], а также, подобно Лиспу, в качестве инструментальной среды для реализации языков представления знаний [Грох и др., 1978; Сергиевский, 1986; Хорошевский, 1986].

В начале 70-х годов появился новый язык, способный, по-видимому, составить конкуренцию Лиспу при реализации систем, ориентированных на знания, — Пролог [Клоксин и др., 1987; Colmegeauer, 1983]. Этот язык не дает новых сверхмощных средств программирования по сравнению с Лиспом, но поддерживает другую модель организации вычислений. С математической точки зрения Пролог — язык хорновских дизъюнктов [Kowalski, 1979]. С практической точки зрения привлекательность его состоит в том, что, подобно тому, как Лисп скрыл от программиста устройство памяти ЭВМ, Пролог позволил ему не заботиться (без необходимости, конечно) о потоке управления в программе.

Пролог — европейский язык, был разработан в Марсельском университете в 1971 г. Однако популярность он стал приобретать лишь в начале 80-х годов. И связано это с двумя обстоятельствами: во-первых, благодаря усилиям математиков был обоснован логический базис этого языка [Минц, 1986] и, во-вторых, в японском проекте вычислительных систем пятого поколения [Мото-ока, 1984] он выбран был в качестве базового для одной из центральных компонент — машины вывода. В настоящее время Пролог начинает завоевывать признание и на американском континенте, хотя, безусловно, уступает в популярности Лиспу и даже специальным продукционным языкам.

Среди языков, с появлением которых возникали новые представления о реализации интеллектуальных систем, необходимо выделить языки, ориентированные на программирование поисковых задач. Это Плэнер [Пильщиков, 1983] и различные его модификации [Sussman et al., 1971; Пильщиков, 1982], Коннайвер [McDermott, 1972], а также языки, выросшие из потребностей известной планирующей системы QA4 [Sacredoti, 1976; Derksen, 1973]. Следует сразу отметить, что все эти языки функционируют в Лисп-среде и создавались как расширения базового языка. Для них кроме свойств Лиспа, характерны следующие черты: представление данных в виде произвольных списковых структур; развитие методы сопоставления образцов; поиск с возвратами и вызов процедур по образцу. Ни один из этих языков не стал универсальным языком программирования искусственного интеллекта, подобным Лиспу. Однако выработанные здесь решения были использованы и в Лиспе, и в Прологе, и в современных продукционных языках. Важно и то, что языки этой группы способствовали переосмыслению самого понятия программы. В области искусственного интеллекта это послужило толчком к развитию объектно-ориентированного программирования и разработке языков представления знаний первого поколения.

Уже накоплен определенный опыт реализации интеллектуальных систем и можно говорить о том, что в области искусственного интеллекта сформировались определенные парадигмы программирования. Основными из них, по-видимому, являются функциональная, логическая и объектно-ориентированная парадигмы. В последнее время на этот же уровень выходит и продукционное программирование.

*Функциональная парадигма* — программа состоит из независимых функций, каждая из которых определяет правило преобразования своих аргументов для получения некоторого результата. Эти функции определяются путем композиции системных или определяемых программистом функций с использованием структур типа альтернативы или рекурсии. Наиболее известным представителем языков, опирающихся на функциональную парадигму, является Лисп.

*Логическая парадигма* основана на использовании механизма доказательства теорем, который позволяет выяснить, противоречиво ли некоторое множество логических формул. При этом программа может рассматриваться как набор ло-

гических формул совместно с теоремой (запросом), которая должна быть доказана. Формулы ограничены импликантами, а каждая часть импликанты ограничена логическим выражением. Формулы предназначены для представления различных фактов (данных) и правил вывода. Для хранения и эффективной работы с фактами и правилами привлекаются средства баз данных и сопоставление с образом (унификация). При этом программист избавляется от необходимости определения точной последовательности шагов для выполнения вычислений. Типичным представителем этой парадигмы является язык Пролог.

*Объектно-ориентированная парадигма* — программа состоит из объектов и сообщений. Такой способ программирования вносит в программу модульность посредством абстракции данных и наследования и особенно хорошо подходит для ситуаций, когда имеется ясная иерархическая классификация объектов. Это позволяет избежать дублирования и локализовать описания механизмов работы с информацией. Одним из первых языков этого типа является Smalltalk.

Конечно, в современной практике интеллектуального программирования отдельные парадигмы редко используются в чистом виде. И в качестве примеров слияния различных парадигм можно привести не только отдельные языки, но и целый подход к программированию интеллектуальных систем на базе *продукционной парадигмы*, в рамках которой взаимодействие правил-продукций может опираться на функциональную парадигму, а унификация условий применимости правил — на логическую. При этом сам продукционный процессор может быть построен на базе механизмов асинхронного управления, близких по своей идее к объектно-ориентированной парадигме. Таким образом, современный этап программирования интеллектуальных систем характеризуется тенденцией смешанного использования разных парадигм.

Анализ существующих языков обработки символической информации, использование их для реализации интеллектуальных систем, а также сравнение тенденций развития этих языков позволяют сделать несколько замечаний.

1. Можно предположить, что Лисп еще значительное время будет оставаться основным языком для реализации интеллектуальных систем.

2. Уже в ближайшее время можно ожидать появления языков, вобравших в себя лучшие черты Лиспа и, например, Пролога.

3. Наблюдается явная тенденция к созданию параллельных версий языков обработки символической информации и языков для программирования задач искусственного интеллекта.

4. По-видимому, наиболее быстрыми темпами в разработку систем, основанных на знаниях, будет входить объектно-ориентированная парадигма и, как следствие, языки типа Smalltalk. Вместе с тем можно предположить, что Лисп (как это уже было в прошлом) попытается совершить еще одну мутацию и за счет этого вернуть лидирующие позиции.

5. Языки типа Лисп, Пролог, Рефал (а также всевозможные модификации и «смеси» этих и/или других языков символической обработки) будут все больше уступать свои позиции на уровне инженеров по знаниям специальным языкам представления знаний, оставаясь инструментарием системных программистов.

### 1.3. Язык Лисп и его модификации

О. В. Ковригин, К. Г. Перфильев

Язык Лисп — один из наиболее распространенных базовых языков систем искусственного интеллекта. Популярность его [Charniak et al., 1980] объясняется многими причинами. Язык Лисп является одним из немногих языков, ориентированных на работу с символической информацией, а процесс решения большинства задач искусственного интеллекта сводится к обработке такой информации. Кроме того, Лисп представляет собой интерпретирующую систему, а это позволяет значительно облегчить и ускорить процесс создания сложных комплексов программ в интерактивном режиме, так как обеспечивает немедленную реакцию системы

на изменения, вносимые пользователем, и предоставляет мощные средства отладки и редактирования программ. Идеология языка Лисп крайне проста: данные и программы представляются в нем в одной и той же форме. Благодаря такой унификации представления данные могут интерпретироваться как программа, а любая программа может быть использована в качестве данных любой другой программой. Программирование на языке Лисп значительно облегчается и вследствие того, что пользователь избавлен от необходимости следить за распределением памяти: эту функцию выполняет интерпретатор. Язык Лисп является языком функционального программирования. Применение функциональных языков открывает широкие перспективы, позволяя пользователю описывать скорее природу своих задач, чем способ их решения.

Одним из основных недостатков Лиспа традиционно считалась относительно невысокая скорость исполнения программ. Однако с появлением мощных Лисп-машин и с разработкой эффективных компиляторов с языка Лисп скорость исполнения программ значительно увеличилась. Относительное неудобство в освоении языка Лисп состоит в том, что существует много диалектов и, к сожалению, ни один из них не принят в качестве стандарта. Однако сейчас есть надежда, что таким стандартом станет диалект Common Lisp.

Язык Лисп входит сегодня в программное обеспечение почти всех ЭВМ, выпускаемых за рубежом. Большие компьютеры IBM снабжаются интерпретаторами и компиляторами с диалектов Interlisp и Common Lisp, компьютеры DEC — системами Franz Lisp, Interlisp, Xlisp. На ПЭВМ наибольшее распространение получили системы Golden Common Lisp, muLisp, IQ-Lisp. Появились и отечественные Лисп-системы для ЕС ЭВМ. Несмотря на различие диалектов, основными из которых являются Common Lisp, Interlisp, Standard Lisp, Franz Lisp, muLisp [Guy, 1984; Urmi, 1976; Marti et al., 1979; Foderaro et al., 1983; muLISP, 1985], идеология их построения одинакова [Allen, 1978]. Исходя из этого, рассмотрим общие принципы семейства диалектов Лисп, обращая внимание, где это необходимо, на различия.

### Основные понятия

Основная структура данных в Лисп — это *S-выражение* (от Symbolic — символичный), которое определяется как число, литеральный атом, строка, список.

*Литеральный атом* — это последовательность букв и цифр. Строго говоря, числа в Лиспе также являются атомами. Примерами литеральных атомов могут выступать символы: ЛИСП, ФУНКЦИОНАЛЬНЫЙ, ЯЗЫК. Максимальная длина литеральных атомов зависит от конкретной реализации языка (в некоторых диалектах литеральный атом должен обязательно начинаться с буквы, но это не так, например, для muLISP, где он может начинаться и с цифры). В языке Лисп определены два стандартных атома T и NIL, выполняющих роль понятий ИСТИНА и ЛОЖЬ.

*Строка* — последовательность букв и цифр, начинающаяся и заканчивающаяся двойными кавычками « ». Строки отличаются от атомов тем, что могут содержать в качестве элементов разделители атомов: пробелы, точки, запятые. Строка «ЛИСП — ФУНКЦИОНАЛЬНЫЙ ЯЗЫК» представляет собой атом, печатное имя которого состоит из 26 символов, включающих 3 пробела.

Фундаментальным понятием в языке Лисп является понятие списка. Даже название LISP образовано от LIST Processing — обработка списков. *Списком* называется конструкция, состоящая из левой круглой скобки, последовательности S-выражений, каждое из которых может быть либо атомом (литеральным, или числовым, или строкой), либо списком, и правой круглой скобки. Так, список (ФУНКЦИОНАЛЬНЫЙ ЯЗЫК ЛИСП) состоит из трех атомов, список (S-ВЫРАЖЕНИЕ (ОПРЕДЕЛЯЕТСЯ (РЕКУРСИВНО))) — из двух элементов: атома S-ВЫРАЖЕНИЕ и списка (ОПРЕДЕЛЯЕТСЯ (РЕКУРСИВНО)), который, в свою очередь, состоит из атома ОПРЕДЕЛЯЕТСЯ и списка (РЕКУРСИВНО). Важно выделить понятие *пустого списка* — списка, который состоит из  $\emptyset$  элементов, он обозначается ( ) или атомом NIL. Один из парадоксов Лиспа состоит в том, что пустой список — это атом.

Список представляет собой *ссылочную структуру*. Основная *ссылочная структура* языка — так называемая *точечная пара*, которая состоит из указателей на первый и второй элементы пары. Так, *точечная пара* с указателями на атомы А и В изображается в символьной нотации как (А.В). Точечные пары своими указателями могут ссылаться не только на атомы, но и на другие точечные пары. Если *точечная пара* первым указателем ссылается на атом А, а вторым — на другую *точечную пару* (В.С), то это изображается в символьной нотации как (А.(В.С)). Для удобства чтения списков принято соглашение, по которому список, следующий после точки, теряет свои внешние скобки, а точка опускается; если после точки следует атом NIL, то точка и NIL опускаются. Следовательно, последняя запись приводится к виду (А В.С). Часто в литературе используется и другое изображение списков — графическое. Оно более наглядно передает *ссылочную структуру* списков. В графической нотации легко, например, изобразить циклический список, который можно организовать средствами Лиспа, но в символьной нотации изобразить невозможно. Без полного понимания *ссылочной структуры* списков Лиспа невозможно понимание всей стройной организации этого языка.

Любая Лисп-система представляет собой небольшую программу, назначение которой — выполнение программ путем интерпретации S-выражений, подаваемых на вход. Механизм работы Лисп-системы крайне прост. Он состоит из трех последовательных шагов: считывание S-выражения (READ); интерпретация S-выражения (EVAL); печать S-выражения (PRINT).

Синтаксис программ тоже прост — любое S-выражение представляет собой синтаксически правильную программу. С точки зрения синтаксиса программы и данные не различаются. И, следовательно, любое S-выражение Лиспа можно попытаться интерпретировать как программу. Интерпретация S-выражений — это единственная и главная задача Лисп-интерпретатора. Ее выполняет программа Лисп-системы EVAL (точнее, функция EVAL). Однако не любое S-выражение может быть удачно интерпретировано из-за семантических соображений. Функция EVAL как раз и определяет семантику различных диалектов Лиспа — берет одно S-выражение, интерпретирует его, если это возможно, и возвращает другое S-выражение, которое является результатом интерпретации первого S-выражения. Несмотря на большое разнообразие диалектов, можно выделить некоторые общие положения, реализуемые функцией EVAL при интерпретации S-выражений:

если S-выражение является числом или атомом Т или NIL, то EVAL возвращает это S-выражение без изменений;

если S-выражение является литеральным атомом, то функция EVAL возвращает последнее значение, которое было присвоено этому атому, в противном случае большинство интерпретаторов дает сообщение об ошибке;

если S-выражение представляет собой список вида (f arg1 arg2 ... argN), то функция EVAL пытается интерпретировать его следующим образом: первый элемент списка интерпретируется как имя функции, которую необходимо выполнить, взяв в качестве аргументов оставшиеся элементы списка arg1, arg2, ..., argN. В случае удачи функция EVAL возвращает S-выражение, являющееся результатом выполнения функции f. Чтобы функция EVAL могла интерпретировать первый атом в S-выражении вида (f arg1 arg2 ... argN) как имя функции, необходимо, чтобы функция с таким именем была определена к моменту интерпретации.

Все Лисп-системы имеют некоторый набор базовых функций, которые изначально встроены в интерпретатор. Кроме того, пользователь может определять свои собственные функции на языке Лисп, используя специальные конструкторы функций. Если атом f не удается интерпретировать как встроенную функцию языка или как функцию, определенную пользователем, большинство интерпретаторов выдают сообщение об ошибке.

Множества базовых функций различных диалектов сильно отличаются друг от друга, и их число колеблется от нескольких десятков до нескольких сотен. Встроенные функции выполняются с большей скоростью, чем функции, определенные пользователем, так как первые реализованы на том же языке, что и вся

Лисп-система (Ассемблер, Фортан, С, Паскаль). Чрезмерное увеличение базового набора функций приводит к уменьшению рабочей памяти, отводимой под задачи пользователя.

### Стандартные функции

**Функции работы со списками.** Поскольку Лисп — язык обработки списков, то в нем предусмотрено большое количество функций, предназначенных для создания и изменения списков, а также для доступа к элементам списка.

Для создания точечной пары (элементарного ядра любого списка) определена функция CONS — функция двух аргументов, вычисляемых перед обращением, т. е. к ним, в свою очередь, применяется функция EVAL:

$$(CONS 'A 'B) \Rightarrow (A . B)$$

Знак ' означает отмену интерпретации S-выражения, перед которым он стоит. Конструктор CONS можно использовать и для построения более сложных, чем точечная пара, списковых структур:

$$\begin{aligned} CONS 'A '(B) &\Rightarrow (A B) \\ (CONS 'A '(B C)) &\Rightarrow (A B C) \\ (CONS '(A) '(B C)) &\Rightarrow ((A) (B C)) \end{aligned}$$

LIST — функция нефиксированного числа аргументов, из которых она строит новый список:

$$\begin{aligned} (LIST 'LISP) &\Rightarrow (LISP) \\ (LIST 'LISP 'ЭТО 'ЯЗЫК) &\Rightarrow (LISP \text{ ЭТО ЯЗЫК}) \\ (LIST '(A) '(B C)) &\Rightarrow ((A) (B C)) \end{aligned}$$

Функция APPEND берет два аргумента, которые должны быть списками, и строит из них новый список:

$$(APPEND '(LISP \text{ ЭТО}) '(ЯЗЫК)) \Rightarrow (LISP \text{ ЭТО ЯЗЫК}).$$

**Функции доступа к элементам списка.** Основными функциями доступа к элементам списка являются встроенные функции CAR и CDR. Это функции одного аргумента, вычисляемого перед обращением к функции.

Функции CAR возвращает в качестве значения первый элемент списка. Если в качестве аргумента функции CAR выступает литеральный атом, то результат исполнения функции зависит от конкретного диалекта. В одних (muLisp, InterLisp, Lisp F3) применение CAR к атому разрешено, в других (например, Common Lisp, Standard Lisp) это вызовет сообщение об ошибке. Такая операция эквивалентна применению функции EVAL к атому. Во многих диалектах для удобства определены функции CAAR, CAAAR и т. п. Их применение равнозначно последовательному применению функции CAR столько раз, сколько букв А в имени функции.

Функция CDR предназначена для исключения из списка первого элемента. Для одних диалектов применение CDR к литеральному атому недопустимо, в других — функция CDR от атома возвращает список свойств этого атома.

Используя функции CAR и CDR, можно осуществить доступ к любому элементу списка. Во многих диалектах дополнительно имеется функция NTH, выбирающая элемент по указанному номеру.

**Функции изменения списков.** Функция RPLACA от двух аргументов (RPLACA X Y) меняет ссылку первой точечной пары списка X на S-выражение Y:

$$\begin{aligned} (RPLACA '(LISP \text{ ЯЗЫК ИИ}) 'ПРОЛОГ) &\Rightarrow (ПРОЛОГ \text{ ЯЗЫК ИИ}) \\ (RPLACA '(LISP \text{ ЯЗЫК ИИ}) '(LISP \text{ И ПРОЛОГ})) &\Rightarrow \\ &\Rightarrow ((LISP \text{ И ПРОЛОГ}) \text{ ЯЗЫК ИИ}) \end{aligned}$$

Функция RPLACD меняет вторую ссылку точечной пары на новое S-выражение:

$$\begin{aligned} (RPLACD '(LISP \text{ ЭТО ЯЗЫК}) '(ЭТО \text{ ИНТЕРПРЕТАТОР})) &\Rightarrow \\ &\Rightarrow (LISP \text{ ЭТО ИНТЕРПРЕТАТОР}) \end{aligned}$$

Для проведения операций над списками определены функции REMOVE и SUBST. Первая функция служит для удаления элементов из списка, а вторая — для замены элементов. В современных диалектах языка Лисп функции REMOVE и SUBST реализуются в условном варианте. Это означает, что добавляется еще один аргумент, являющийся предикатом, истинность которого проверяется перед применением функции на элементах изменяемого списка. Если результат проверки — истина, то функция выполняется, в противном случае нет. Полезной также представляется функция REVERSE, переупорядочивающая элементы списка в обратном направлении.

**Аппарат реализации функций.** Основан на лямбда-исчислении Черча [Хендерсон, 1982]. Лямбда-выражением называется S-выражение вида

(LAMBDA (arg1 arg2 ... argN) expr1 expr2 ... exprN)

где LAMBDA — специальный атом, зарезервированный в языке для определения функций; arg1, arg2, ... argN — атомы, используемые для указания формальных аргументов функции, определяемой лямбда-выражением; а expr1, expr2, ... ..., exprN — любые S-выражения, составляющие тело функции. Так, например, S-выражение (LAMBDA (X Y) (CONS X Y)) задает функцию от двух аргументов X и Y, а тело функции составляет стандартная функция CONS, образующая новый список.

**Лямбда-конверсией** называется процесс связывания формальных переменных функции с фактическими значениями и интерпретация тела функции. В качестве результата вычисления лямбда-выражения (т. е. значения функции) возвращается значение последнего S-выражения, составляющего тело функции. Например, процесс интерпретации S-выражения ((LAMBDA (X Y) (CONS X Y)) 'A 'B) происходит согласно приведенным выше правилам. Первый элемент выражения (LAMBDA (X Y) (CONS X Y)) представляет собой определение функции, и интерпретатор применяет ее к оставшимся элементам выражения (A и B) как к аргументам. На этом этапе происходит связывание формальных аргументов функции с фактическими значениями, X связывается с A, а Y с B. При каждом обращении к функции формируется стек функциональных связей. После этого интерпретируется тело функции (CONS A, B). Результатом выполнения лямбда-выражения будет список (A . B). После выполнения функции формальные аргументы функции освобождаются от связей.

**Лямбда-определение** функции предполагает, что перед связыванием формальных аргументов с фактическими к фактическим должна быть применена функция EVAL, т. е. они должны быть оценены. Так, если атому A ранее было присвоено значение 1, а атому B — значение 2, то обращение функции ((LAMBDA (X Y) (CONS X Y)) A B) приведет сначала к вычислению A и B, а затем только произойдет связывание X с 1, а Y с 2. И результатом вычисления функции будет пара (1 . 2). Во многих диалектах Лиспа (InterLisp, Lisp F3 и muLisp) есть возможность определять функции, которые не оценивают свои фактические аргументы перед связыванием. Для этого вводится специальная форма лямбда-выражений NLAMBDA. С помощью NLAMBDA обеспечивается передача параметров функции не по значению, а по имени.

Для формирования функций с нефиксированным (произвольным) числом аргументов применяется тот же аппарат LAMBDA и NLAMBDA с той лишь разницей, что на месте списка аргументов находится один атом. В этом случае при обращении к функции этот единственный аргумент связывается со списком фактических аргументов.

### Функции, определяемые пользователем

С помощью аппарата лямбда-выражений в Лиспе можно определить поименованные функции. Для этой цели необходим *конструктор функций*, осуществляющий связывание некоторого атома (именн функции) с лямбда-определением функции. В качестве таких конструкторов, как правило, выступают функции PUTD и DEFUN. Функция PUTD от двух аргументов связывает атом, являю-



щийся первым аргументом, с LAMBDA- или NLAMBDA-определением, выступающим в качестве второго аргумента, а возвращает в качестве своего значения имя определяемой функции:

$(PUTD\ 'FOO\ '(LAMBDA\ (X\ Y)\ (CONS\ X\ Y))) \Rightarrow FOO.$

Многие функции в Лиспе определяются и используются не столько для получения значения функции, которое оно возвращает, сколько для выполнения побочных действий: связывания определений, присвоения глобальных значений, определения структур, ввода и вывода информации. Типичным представителем таких функций является и функция PUTD, основное назначение которой не вернуть имя функции, а связать его с некоторым определением. Так, определив новую функцию FOO, мы можем обращаться к ней следующим образом:  $(FOO\ 'A\ 'B) \Rightarrow (A\ .\ B)$ . Для удобства записи вводится функция DEFUN, имеющая тот же побочный эффект, что и PUTD:

$(DEFUN\ FOO\ (X\ Y)\ (CONS\ X\ Y)) \Rightarrow FOO$

Отметим, что в теле функции могут быть использованы и свободные (несвязанные) переменные, не указываемые в списке аргументов и выступающие в качестве глобальных имен для данной функции. Перед интерпретацией тела функции такие свободные переменные должны быть определены, т. е. иметь значения. Для доступа к определениям предназначена функция GETD одного аргумента (имя функции), которая возвращает лямбда-определение данной функции:

$(GETD\ 'FOO) \Rightarrow (LAMBDA\ (X\ Y)\ (CONS\ X\ Y))$

Любой литеральный атом может иметь значение, представляющее собой S-выражение. Различают *глобальные* и *локальные значения атомов*. Последние могут связываться с атомом на время выполнения некоторых функций. Для установления значений атома используются функции SET, SETQ и SETQQ. Последние два являются разновидностью первой с той разницей, что SET оценивает свои аргументы перед связыванием, SETQ не вычисляет первый аргумент, а функция SETQQ вообще не оценивает своих аргументов. Указанные функции имеют два аргумента, значением первого является литеральный атом (кроме T и NIL), а значением второго — произвольное S-выражение. Обращение к функции  $(SETQ\ X\ 'LISP)$  приводит к побочному эффекту, в результате атом X получает значение LISP. В качестве значения функции возвращается второй ее аргумент.

Глобальные значения атомов могут быть установлены с помощью указанных функций. Тогда, если атом, имеющий глобальное значение, выступает в качестве формального аргумента какой-либо функции, то при лямбда-конверсии он связывается с новым значением. После выполнения функции ему возвращается старое глобальное значение.

**Предикативные функции.** Значением предикативных функций может быть либо атом NIL, обозначающий ЛОЖЬ, либо любой другой атом, отличный от NIL (чаще всего атом T) и обозначающий ИСТИНУ. Основными предикатами в языке Лисп являются ATOM, LISTP, NUMBERP, NULL, MEMBER, EQ, EQUAL. Предикаты ATOM, LISTP и NUMBERP определены для одного аргумента и возвращают значение T, если S-выражение — атом, список, число. Предикат NULL возвращает значение T, если его аргумент — атом NIL или пустой список ( ).

Для сравнения S-выражений на различных уровнях представления используются предикаты EQ и EQUAL. Функция EQ служит для установления идентичности двух S-выражений. Два выражения считаются идентичными, если они занимают одно и то же место физической памяти в Лисп-системе. Функция EQUAL сравнивает два S-выражения и возвращает значение T, если S-выражения одинаковы, но, возможно, занимают разную физическую память.

К предикативным функциям можно отнести функции OR, AND и COND. Они служат для построения сложных условных выражений в Лиспе. Аргументами функций OR и AND (число аргументов не фиксировано) должны быть предика-

тивные функции. Обращение к функциям имеет следующий вид:

(OR pred1 pred2 ... predN) (AND pred1 pred2 ... predN).

Интерпретация функции OR состоит в последовательном вычислении предикатов до тех пор, пока не будет использован предикат, имеющий значение Т. Интерпретация функции AND состоит в последовательном вычислении предикатов до тех пор, пока не будет использован предикат, имеющий значение NIL.

В качестве условного управления процессом интерпретации функций может использоваться и функция COND:

(COND (pred1 exp1.1 exp1.2 ... exp1.M)  
(pred2 exp2.1 exp2.2 ... exp2.K)

(predN expN.1 expN.2 ... expN.L)),

где pred1, pred2, predN — предикативные функции, а exp1.1, ..., expN.L — любые S-выражения. При интерпретации COND-выражения отскакивается первая альтернатива, у которой результат оценки первого элемента (предикативная функция) — Т, в этом случае выполняются S-выражения, следующие за предикатом, а дальнейшая оценка лар прекращается.

**Рекурсия и итерация.** Лисп — типичный и самый распространенный язык функционального программирования, основанный на широком использовании эффективных механизмов рекурсии в процессе вычислений. Это и является одним из главных его отличий от традиционных языков. Вместе с тем для обеспечения «идеологической» совместимости с другими языками программирования, а также для повышения эффективности программ при решении некоторых частных задач в язык была введена группа функций, предоставляющих возможность организации итерационной обработки информации.

В указанной группе прежде всего выделяются так называемые MAP-функции: MAPC, MAPCAR, MAPLIST и др. Каждая из них имеет более двух аргументов, значением первого должно быть имя определенной ранее или базовой функции, или лямбда-выражение, вызываемое MAP-функцией итерационно, а остальные аргументы служат для задания аргументов на каждой итерации. Естественно, что количество аргументов в обращении к MAP-функции должно быть согласовано с предусмотренным количеством аргументов у аргумента-функции. Различие между всеми MAP-функциями состоит в правилах формирования возвращаемого значения и механизме выбора аргументов итерирующей функции на каждом шаге. В частности, функции MAPC и MAPCAR в качестве аргументов на первой итерации цикла используют первые элементы соответствующих списков аргументов, на второй — вторые и т. д., а функция MAPLIST — полные списки, заданные при обращении, списки после применения функции CDR и т. д. Функция MAPC возвращает в качестве результата первый из списка аргументов, заданных при обращении к ней, или NIL в некоторых диалектах. Функции MAPCAR и MAPLIST возвращают список результатов применения функции на каждой итерации.

Другим типичным представителем группы итерационных функций может служить функция LOOP, имеющая в общем случае вид (LOOP expr1 expr2 ... exprN), где в качестве аргументов могут быть использованы любые синтаксически и семантически допустимые S-выражения (обращения к функциям, лямбда-выражения) либо специальные конструкции вида ((e1 ... eK)s1 ... sL). В последнем случае указанный список используется в качестве точки анализа условия окончания цикла. Если вычисление S-выражения дает значение NIL, итерационный процесс продолжается, в противном случае прекращается, но предварительно вычисляются последовательно S-выражения e1, ..., eN и последнее из полученных значений возвращается в качестве значений всей функции LOOP.

**Функции интерпретации выражения.** Почти во всех диалектах определены функции APPLY и FUNCALL, позволяющие интерпретировать S-выражения. Обращения к этим функциям имеют следующий вид:

(APPLY fun arg)  
(FUNCALL fun arg1 arg2 ... argN)

APPLY и FUNCALL вычисляют функции, являющиеся их первыми аргументами, производя связывание формальных аргументов с указанными S-выражениями arg или arg1, arg2, ..., argN. В качестве значения возвращается результат применения функции fun, которая может быть встроенной или определенной функцией или лямбда-выражением.

Необходимо отметить еще одну особенность языка Лисп, которая вытекает из природы организации структур данных и программ и механизма их интерпретации. На языке легко реализовывать задачи автоматического синтеза программ. Он позволяет с помощью одних функций формировать определения других функций, программно анализировать и редактировать эти определения как S-выражения, а затем, используя функции типа EVAL, исполнять их.

**Атомы и списки свойств, ассоциативные списки.** Наряду со значением любой литеральный атом в языке Лисп может иметь специальный присоединенный список, называемый *списком свойств*. Структура этого списка такова:

(prop1 val1 prop2 val2 ... propN valN)

где prop1, prop2, ..., propN — литеральные атомы, обозначающие имена свойств, а val1, val2, ..., valN — S-выражения, соответствующие значениям этих свойств.

Список свойств атома формируется с помощью специальной функции PUT (или PUTP):

(PUT atom prop val) ⇒ val

где atom — имя атома, у которого формируется список свойств; prop — имя свойства, а val — значение.

Для доступа к свойствам атомов определена функция GET (или GETP) вида (GET atom prop), которая возвращает значение свойства для данного атома (или для Common Lisp список, состоящий из имени свойства и его значения). Для изменения списка свойств предусмотрены функции REMPROP и ADDPROP. Первая позволяет удалить указанное свойство с его значением, вторая модифицирует значение заданного свойства.

Введение в язык Лисп средств поддержки атомов со списком свойств способствовало эффективной реализации концепции фреймов и их сетей. Некоторые диалекты языка реализуют список свойств таким образом, что получить его можно с помощью обращения к функциям CDR. Аналогичные структуры данных могут быть реализованы с помощью ассоциативных списков.

*Ассоциативным списком* в одних диалектах языка Лисп называют список вида

((key1 val1) (key2 val2) ... (keyN valN))

а в других —

((key1. val1) (key2. val2) ... (keyN. valN))

Поскольку в языке отсутствуют специальные физические средства организации и хранения ассоциативных списков, они реализуются как значения атомов; для работы с ними предусмотрены только две функции выборки ASSOC и RASSOC, позволяющие получить доступ к элементам ассоциативного списка по ключу или по значению:

(SETQ X '((JAPAN TOKIO) (USSR MOSCOW))) ⇒  
 ⇒ ((JAPAN TOKIO) (USSR MOSCOW))  
 (ASSOC X 'USSR) ⇒ (USSR MOSCOW)  
 (RASSOC X 'TOKIO) ⇒ (JAPAN TOKIO)  
 (ASSOC X 'USA) ⇒ NIL

### Макросредства

Использование макросредств, предлагаемых современными Лисп-системами, — один из самых эффективных путей реализации сложных программ. При наличии макросредств некоторые функции в языке могут быть определены в виде макрофункций. Такое определение фактически задает закон предварительного

построения тела функции непосредственно перед фазой интерпретации. Интерпретация функций, определенных как макро, производится в два этапа. На первом, называемом *макрорасширением*, происходит формирование лямбда-определения функции в зависимости от текущего контекста, на втором осуществляется интерпретация созданного лямбда-выражения. Использование макрофункций облегчает построение языка с листоподобной структурой, имеющего свой синтаксис, более удобный для пользователя. С другой стороны, макроопределения позволяют реализовать функции типа NLAMBDA в тех диалектах, где этот тип функций отсутствует.

Одной из разновидностей макроопределений является READ-макро. Этот аппарат позволяет изменять реакцию функции READ при вводе каких-либо специальных символов. Примерами READ-макро могут выступать средства автоматического декодирования знака в обращение к функции QUOTE и знака комментария «;». Однако чрезмерное использование макросредств затрудняет чтение и понимание программ.

### Функция ввода-вывода

Современные диалекты языка Лисп, как правило, имеют развитые средства управления вводом-выводом. Основу этих средств составляют три основные функции READ, RATOM и PRINT. Первые две позволяют осуществлять операции ввода S-выражений (READ) и атомов (RATOM), последняя выполняет вывод S-выражений.

Базовый набор функций обычно наряду с указанными функциями включает различные их модификации и дополнительные функции (типа TERPRI, SPACES), позволяющие при программировании легко получить некоторые дополнительные эффекты (автоматическое дополнение печатной строки с изображением S-выражения специальными символами перевода каретки на начало следующей строки, блокировка выделяющих метасимволов при выводе литеральных атомов, в печатных именах которых присутствуют непечатные символы и т. д.). Кроме того, практически каждый диалект содержит набор функций управления входными и выходными потоками для связи с внешними устройствами ЭВМ. Однако указанные функции являются одной из наиболее машинно-зависимых составляющих Лисп-систем, поскольку по необходимости учитывают специфику среды операционной системы.

### Организация памяти и режимы работы Лисп-систем

В основе всех существующих стратегий организации памяти Лисп-систем лежат следующие принципы:

управление памятью, включая операции распределения и очистки, производится Лисп-системой автоматически без участия пользователя;

вся память Лисп-системы подразделяется на три основные части: атомы, списки, числа; с точки зрения хранения для данных и определений функций не делается никакого различия.

При выполнении программ пользователя типичной является ситуация, когда часть памяти Лисп-системы оказывается занятой ненужной информацией вследствие многократных обращений к функциям и создания стеков аргументов, организации временных списков и переменных и т. п. Если свободная память при этом оказывается исчерпанной, возникает необходимость очистки памяти — сборки мусора. В современных Лисп-системах выполнение операции сборки мусора занимает от одной до нескольких секунд. В задачах большого объема сборки мусора запускается весьма часто, что резко ухудшает временные характеристики прикладных программ.

Для повышения быстродействия прикладного программного обеспечения в Лисп-системах используют два основных пути: применение специального D-кода для внутреннего представления списков и замена режима интерпретации режимом компиляции. Последний режим предусмотрен далеко не во всех Лисп-системах, его использование накладывает ряд дополнительных ограничений на технику программирования.

**Common Lisp.** Этот диалект [Guy et al., 1984] отличается наиболее широким представлением различных структур данных и включает около 800 встроенных функций. В этом диалекте обеспечиваются средства обработки основных классов числовой информации: целых (с разделением на подклассы длинного и короткого формата), вещественных (с выделением подклассов короткого и длинного формата, а также двух промежуточных по обеспечиваемой точности вычислений форматов чисел с одинарной и двойной точностью) и комплексных (путем указания координат точки на комплексной плоскости). Символьные данные (литеры, литеральные атомы, строки) в Common Lisp соответствуют основным возможностям других Лисп-систем. Дополнительно имеются средства обработки непечатаемых литер в символьных именах.

Одним из существенных преимуществ диалекта Common Lisp является наличие средств обработки массивов и структур, по своим возможностям не уступающих соответствующим средствам традиционных языков программирования Фортран и Паскаль. Массивы в описываемом диалекте могут иметь любое неотрицательное число измерений и индексируются последовательностью целых чисел. Тип компонентов массива может быть произвольным. Выделяется подкласс векторов — одномерных массивов, среди которых отдельно рассматриваются литерные векторы (строки) и битовые массивы.

Структуры Common Lisp являются типом многокомпонентных записей, определяемых пользователем и имеющих именованные компоненты. Встроенное макроопределение DEFSTRUCT используется для определения структур новых типов. Для создания данных нового типа в виде структуры предусмотрены средства автоматической генерации набора функций, обеспечивающих средства манипулирования объектами этого класса (создание конкретных переменных типа структуры, доступ к элементам структур и т. д.).

Удобным средством контроля доступа к различным переменным и функциям Лисп-программ в диалекте Common Lisp являются пакеты. Каждый пакет представляет собой множество имен объектов, определенных и доступных в нем. Кроме того, внутри пакета имена объектов подразделяются на внутренние и внешние. Первые предназначены для использования только внутри данного пакета, а вторые — для обеспечения связи с другими пакетами. Лисп-интерпретатор предоставляет широкий спектр средств манипулирования с пакетами, включая функции создания и привязки к конкретному пакету, поиска пакета, содержащего заданное имя объекта, переименования пакетов и т. д. Как правило, Лисп-система имеет в своем составе четыре стандартных пакета: lisp (содержащий примитивы системы), user (умалчиваемый пакет прикладных программ и данных пользователя), keyword (содержащий ключевые слова всех встроенных функций) и функций, определяемых пользователем, system (зарезервированный для системных целей).

Значительной переработке и расширению в Common Lisp подверглись средства ввода-вывода и передачи информации. Для эффективной организации различных обменов с внешней средой введена концепция потоков, позволяющих осуществлять одно- и (или) двустороннюю передачу информации. Поскольку потоки здесь рассматриваются как один из стандартных типов объектов Лисп-системы, для них предусмотрен набор базовых функций, включающий, например, предикативные. В диалекте различаются символьные и двоичные потоки. В первом случае передача выполняется по байтам, а во втором — целыми числами. Система предусматривает доступ к семи стандартным потокам: поток стандартного ввода, поток стандартного вывода, стандартный поток вывода диагностических сообщений, двусторонний поток диалога с пользователем, двусторонний поток для целей отладки, поток связи с терминалом, поток вывода при трассировке. Кроме указанных стандартных потоков пользователь имеет возможность создавать и использовать собственные потоки.

В дополнение к указанным типам данных Common Lisp имеет ряд специфических классов объектов: хэш-таблицы, обеспечивающие эффективный способ

доступа к данным по ключу; READ-таблицы, обеспечивающие управление обработкой информации, поступающей из входного потока Лисп-системы, и некоторые другие. Такое множество имеющихся в диалекте различных типов данных, с одной стороны, развенчивает существующее ошибочное представление о языке Лисп как о средстве для обработки только символьной информации, а с другой — наличие мощных средств манипулирования типами данных существенно усложняет его.

При создании диалекта Common Lisp авторы предусмотрели существенное расширение средств управления передачей параметров при описании функций и лямбда-выражений. В диалекте все множество формальных аргументов функции или лямбда-выражений может быть разбито с помощью специальных ключевых слов (&OPTIONAL, &REST, &KEY, &AUX) на пять подмножеств: обязательные параметры, необязательные, остаточный параметр, ключевые параметры, дополнительные.

**Golden Common Lisp.** Этот диалект языка Лисп является результатом адаптации Common Lisp к программной среде ПЭВМ серии IBM PC и других семейств, программно совместимых с ней, выполненной фирмой Gold Hill Computers, Inc. (США). Первая реализация этой системы [Golden, 1985] не охватывает, к сожалению, все без исключения возможности своего прототипа. Вместе с тем попытка использовать специфические особенности ПЭВМ, на которой осуществлялась реализация, привела к включению в систему таких дополнительных средств, как доступ к некоторым функциям операционной системы и средств управления окнами на экране, дисплея пользователя.

Среди достоинств этого диалекта следует отметить большой набор различных дополнительных прикладных программ, поставляемых вместе с Лисп-системой. В частности, этот набор включает мощный интерактивный редактор текстов Лисп-программ, имеющий встроенные средства синтаксического контроля S-выражений, а также большую и весьма удобную для начинающих программистов обучающую систему, в основу которой положен материал монографии [Winston et al., 1984]. Особую ценность предлагаемой обучающей системе придает наличие разделов, демонстрирующих методы решения некоторых типичных задач искусственного интеллекта средствами языка Common Lisp.

**muLisp.** Интерпретатор muLisp-85, разработанный фирмой Soft Warehouse Inc. (США) для ПЭВМ серии IBM PC [muLISP, 1985], — удачный вариант реализации диалекта языка, включающий сравнительно ограниченный набор базовых функций (около 260) и оказавшийся вследствие этого более простым для изучения.

По сравнению с Common Lisp диалект muLisp не имеет такого широкого спектра доступных типов данных. В нем обеспечивается работа только с двумя типами числовой информации: целыми числами с любым основанием и рациональными. В диалекте отсутствуют средства работы со структурами, массивами, потоками и другими типами данных, указанная реализация языка Лисп имеет одно существенное преимущество — возможность пополнения базового набора функций путем подключения подпрограмм, написанных на языке ассемблера, что позволило повысить гибкость использования интерпретатора и эффективность прикладного программного обеспечения, создаваемого на его основе. Возможность такого пополнения отсутствует в большинстве других Лисп-систем, являющихся в этом смысле замкнутыми программными продуктами.

Среди других, вероятно, менее существенных, особенностей системы можно указать на реализацию специального механизма, позволяющего не заботиться о присваивании начальных значений литеральным атомам, получающих изначальное значение, равное «печатиному» имени самого атома. Еще одной особенностью диалекта является возможность использования новой синтаксической конструкции «встроенный COND», существенно сокращающей тексты описаний функций пользователя и применяемой при записи тел функций и лямбда-выражений. В общем случае эта конструкция имеет вид

```
(pred1 exp1.1 exp1.2 ... exp1.M)
(pred2 exp2.1 exp2.1 ... exp2.K)
```

expN.1 expN.2 ... expN.L,

что эквивалентно

(COND (pred1 exp1.1 exp 1.2 ... exp1.M)  
(pred2 exp2.1 exp2.1 ... exp2.K)

(T exp N.1 expN.2 ... expN.L)).

Набор базовых функций muLisp-интерпретатора включает ряд функций, обеспечивающих доступ практически ко всем функциям операционной системы ЭВМ через соответствующие прерывания. Наконец, указанная Лисп-система обеспечивается библиотеками Лисп-функций, дополняющими базовый набор функциями, имеющимися в диалектах Common Lisp и InterLisp, что облегчает решение проблемы переносимости исходных текстов программных модулей, а также библиотеками, позволяющими выполнять манипулирование окнами на экране дисплея и обрабатывать управляющие воздействия через устройство типа «мышь». В комплект дополнительного программного обеспечения к интерпретатору входят интерактивный редактор текстов и простая обучающая система, написанные на диалекте языка muLisp.

**Interlisp.** Этот диалект [Urini, 1976] наряду с Common Lisp один из наиболее распространенных, имеет достаточно развитый аппарат представления и манипулирования различными структурами данных, включая массивы, и большой набор базовых функций (около 400). Среди общих особенностей данного варианта языка следует отметить использование для обозначения встроенных функций нетрадиционных имен, используемых в других диалектах языка Лисп, что порой затрудняет перенос готовых программных продуктов на другие диалекты и другие ЭВМ.

В диалекте Interlisp предоставляется возможность определить функции с «полюсфиксированным» числом аргументов. Например, список аргументов в определении функции выглядит следующим образом: (LAMBDA (X Y . Z) expr1 ... ... exprN). Такая запись означает, что первые два аргумента будут связаны, как в случае фиксированного числа аргументов, а остаток фактических значений оформлен в список и связан с аргументом Z. Существенным отличием этого диалекта является и то, что лямбда-определение функции хранится в списке свойств атома, соответствующего имени функции.

Современные диалекты языка Лисп можно рассматривать как мощные интерактивные системы программирования. Это объясняется двумя причинами. Во-первых, сам язык Лисп претерпевает серьезные изменения — развиваются средства языка, предназначенные для обработки нетрадиционных для Лиспа типов данных: массивов, векторов, матриц; появляются некоторые средства управления памятью (пакеты), отсутствующие в Лиспе. Серьезные изменения претерпевают и управляющие структуры. Появились несвойственные природе языка Лисп функции, заимствованные из Фортрана, Алгола, Паскаля и Бейсика: Do, Loop, Goto, Case и пр., позволяющие пользователю, незнакомому с принципами функциональных языков, легко переходить на Лисп. Качество программ снижается, зато возрастает популярность языка. Во-вторых, если на первом этапе развития Лисп-системам была присуща небольшая скорость обработки данных и серьезные ограничения на емкость используемой оперативной памяти, то современные Лисп-системы уже могут соперничать по этим параметрам с такими языками, как С, Паскаль и др. Использование Лисп-машин вообще практически снимает ограничения памяти и быстродействия.

Для ПЭВМ ограничения по памяти и быстродействию все еще остаются существенными. Однако положение далеко не безнадежно. Развитие Лисп-систем для ПЭВМ идет сегодня по трем различным направлениям. Первое связано с увеличением емкости памяти, которая может использоваться Лисп-системой. С этой целью ряд компаний разработал версии языка Golden Common Lisp, использующие расширения оперативной памяти до 8 Мбайт и виртуальную (на жестком диске) память. Второе направление связано с повышением быстродей-

ствия Лисп-систем. Так, многие Лисп-системы, работающие на ЭВМ IBM PC/XT, используют для арифметических операций специальный сопроцессор 8087. Однако наилучшие результаты достигаются на ЭВМ, снабженных 286 или 386 процессорами с добавлением специального Лисп-процессора Hummingboard-386. Третье направление состоит в разработке эффективных компиляторов программ с языка Лисп в традиционные языки (чаще всего в язык С). Примером такого компилятора может служить система Star Sapphire, осуществляющая трансляцию текста программ, реализованных на диалекте Common Lisp, на язык С. Указанная система позволяет получать программы, которые могут использовать до 710 функций языка Лисп и виртуальную память 8 Мбайт. Кроме того, пакет включает библиотеки на языке С, поддерживающие графику и объектно-ориентированный вариант Лиспа.

Положительным нововведением в современные диалекты можно считать псевдоассемблерные команды, которые позволяют оперировать основными регистрами машины и организовывать прерывания на уровне DOS и BIOS. Кроме того, многие Лисп-системы имеют хорошие интерфейсы с другими языками (Фортран, Ассемблер, С, Паскаль), что позволяет повысить эффективность прикладных Лисп-программ.

Если же говорить о глобальной тенденции развития самой идеологии языка Лисп, то очевидно, что она связана с созданием объектно-ориентированных версий языка как наиболее пригодных для реализации систем искусственного интеллекта.

#### **1.4. Язык Пролог и методы его реализации**

*Н. И. Ильинский, И. Б. Козинцев*

Пролог — язык логического программирования (ПРОграммирование в ЛОГике). В основе языка результаты по автоматизации доказательства теорем в исчислении предикатов первого порядка. Пролог-программа состоит из двух частей: базы данных (соответствует множеству аксиом) и последовательности целевых утверждений, описывающих в совокупности отрицание доказываемой теоремы. Главное принципиальное отличие интерпретации программы на Прологе от классической процедуры автоматического доказательства теоремы в исчислении предикатов первого порядка состоит в том, что аксиомы в базе данных упорядочены и порядок их следования весьма существен, так как на этом часто основан сам алгоритм, реализуемый Пролог-программой. Другим существенным ограничением Пролога является то, что в качестве логических аксиом используются формулы ограниченного класса — так называемые дизъюнкты Хорна. Однако при решении многих практических задач этого достаточно для адекватного представления знаний.

Очевидно, что поиск «полезных» для доказательства формул — комбинаторная задача и при увеличении числа аксиом число шагов вывода катастрофически быстро растет. Поэтому в реальных системах применяют всевозможные стратегии, ограничивающие слепой перебор. В языке Пролог реализована стратегия линейной резолюции, предполагающая использование на каждом шаге в качестве одной из сравниваемых формул отрицание теоремы или ее «потомка», а в качестве другой — одну из аксиом. При этом выбор той или иной аксиомы для сравнения может сразу или через несколько шагов завести в «тупик». Это вынуждает вернуться к точке, в которой производился выбор, чтобы испытать новую альтернативу, и т. д. Порядок просмотра альтернативных аксиом не произволен — его задает программист, располагая аксиомы в базе данных в определенном порядке. Кроме того, в Прологе предусмотрены достаточно удобные (а во многих случаях просто необходимые) «встроенные» средства для запрещения возврата в ту или иную точку в зависимости от выполнения определенных условий. Таким образом, процесс доказательства в Прологе существенно более прост и целенаправлен, чем в классическом «недетерминированном» методе резолюций, за счет применения определенных эвристик.



## Основные понятия

Существует несколько вариантов синтаксиса Пролога в зависимости от метода реализации транслятора (интерпретатора или компилятора) для различных ЭВМ и операционных систем. В различных реализациях имена встроенных (системных) предикатов, их набор, а иногда и семантика сильно отличаются друг от друга. Но есть общие черты во всех реализациях. Прежде всего, это общая структура Пролог-программ, которые включают два компонента — базу данных (последовательность аксиом) и конъюнкции, точнее, последовательности целевых утверждений (отрицания теоремы). Аксиомы базы данных объединяются в подмножества, в которых их порядок строго регламентирован, сами же подмножества не упорядочены.

**База данных** — это множество утверждений. Утверждения базы данных бывают двух типов: факты и правила. Факт — аналог аксиомы вида  $T \rightarrow Q$ , где  $T$  — тождественно истинный предикат;  $Q$  — предикат. Для обозначения факта используется запись вида  $\langle Q \rangle$  — предикат, вслед за которым ставится точка. Предикаты бывают 0-местные (без аргументов) или  $n$ -местные ( $n$  больше либо равно 1, с аргументами). Если предикат 0-местный, он обозначается одним именем. Предикат с аргументами обозначается именем, вслед за которым в скобках через запятые записываются все его аргументы. Аргумент предиката — синтаксическая конструкция, называемая *термом*. В отличие от классического исчисления предикатов первого порядка сам предикат в Прологе рассматривается как терм и может выступать в роли аргумента другого предиката. Терм — это либо константа, либо переменная, либо структура. Переменная — это либо знак  $\_$  (подчеркивание), либо последовательность букв и/или цифр, начинающаяся с прописной латинской буквы. Переменная  $\_$  называется анонимной.

**Правило** — аналог аксиомы вида  $P_1 \& \dots \& P_k \rightarrow Q$ , где  $P_i$  (условия) и  $Q$  (следствие) — предикаты. В Прологе обычно применяется инверсная запись правила, заканчивающаяся точкой:

$$Q: \_ P_1, \dots, P_k.$$

которое читается так: «цель  $Q$  удовлетворяется, если удовлетворяются подцели  $P_1$  и  $\dots$  и  $P_k$ » или проще: « $Q$ , если  $P_1$  и  $\dots$  и  $P_k$ ». Предикат  $Q$ , стоящий слева от знака  $\leftarrow$ , называется *заголовком правила*, а предикаты условия составляют *тело правила*.

Особую роль в Прологе играют *целевые утверждения*, для записи которых используется следующая нотация:

$$? \_ R_1, \dots, R_1.$$

При одной и той же базе данных различные целевые утверждения соответствуют различным решаемым задачам. Других объектов, кроме фактов и правил, в базе данных нет.

**Структура** — более сложное, чем константа и переменная, образование, характеризуется именем и компонентами — составными частями структуры. Каждый компонент структуры — это константа, переменная или, в свою очередь, структура. Если структура не содержит переменных ни на каком уровне вложенности подструктур, она называется константной, в противном случае — переменной. Для общности можно считать константы и переменные частным вырожденным случаем константных и переменных структур.

Сами константные структуры, а также отдельные их подструктуры, как элементарные, так и являющиеся константными структурами, могут «кодироваться» с помощью переменных, образуя переменные структуры. При этом «коды» структур могут рассматриваться как некоторые шаблоны или образцы, которые можно сопоставлять исходным структурам или между собой. Необходимость в подобных сопоставлениях возникает в процессе решения исходной задачи интерпретатором Пролога при согласовании целевого утверждения с базой данных, при попытке унификации этого утверждения с каким-либо фактором или с заголовком какого-либо правила. В роли образцов при этом выступают термы, являющиеся аргументами сравниваемых предикатов, имена которых, естественно, должны совпадать. Правила сопоставления с образцами следующие:

1. Если оба сравниваемых образца — константы, для успешного сопоставления они должны совпадать.

2. Если один из сравниваемых образцов — некокретизированная переменная, т. е. переменная, которой не приспано никакого значения, то второй образец может быть любым термом. При этом если этот терм — константа или константная структура, то переменная, выступающая в роли первого образца, конкретизируется — ей приписывается значение, которое является вторым образцом. Если второй образец — другая некокретизированная переменная, то между первой и второй переменными устанавливается прямая связь: в случае дальнейшего означивания одной из них вторая автоматически получает то же значение. Если же второй образец является структурой, содержащей другие переменные, между ними и исходной переменной устанавливается более сложная «функциональная» связь. Так, если сопоставляются переменная  $X$  и структура  $f(Y, Z)$ , дальнейшая конкретизация двух из имеющихся переменных ( $Y, Z$ ) приведет к конкретизации третьей ( $X$ ).

3. Если оба сравниваемых образца — структуры, для их успешного сопоставления должны совпадать имена этих структур, а также сопоставляться между собой все соответствующие компоненты этих структур по приведенным выше правилам. Например, при сопоставлении структур  $f(X, g(X))$  и  $f(a, Y)$  переменная  $X$  получит значение  $a$ , а  $Y = g(a)$ .

4. Конкретизированные переменные при сопоставлении ведут себя так же, как их значения — константы или константные структуры.

По-особому осуществляется сопоставление с образцом, содержащим анонимные переменные  $'\_'$ . Собственно знак  $'\_'$  может изображать физически различные переменные. Так, список  $[-, -, -]$  будет успешно сопоставлен со списком  $[a, b, c]$ , а не только со списками вида  $[a, a, a]$  или  $[b, b, b]$ . Наличие механизма сопоставления с образцом (pattern matching) наряду с механизмом возвратов (backtracking) является характерной особенностью Пролога.

### Методы организации выполнения Пролог-программ

Правила и факты базы данных Пролог-программы можно рассматривать как определение некоторого предиката, который входит в состав целевого утверждения и конкретизирует ту или иную решаемую задачу. Если все аргументы этого предиката представляют собой константы или константные структуры, осуществляется проверка, согласуется ли этот предикат с имеющейся базой данных. Решением задачи является ответ «да» или «нет». Если предикат содержит переменные, осуществляется поиск их значений, для которых имеет место согласование данного предиката с базой данных. Решение задачи должно содержать значения этих переменных.

Часто для управления процессом выполнения Пролог-программ используются два встроенных предиката `cut` и `fail` (отсечение и отказ). Последний — тождественно ложный предикат, выполнение предиката `cut` (обычно он обозначается символом  $\langle ! \rangle$ ) всегда завершается успешно, но сопровождается рядом побочных эффектов. Введение предиката `cut` связано со стремлением получить более эффективные программы, представляя возможность сокращения дерева поиска решения за счет отсечения бесполезных ветвей. Его можно интерпретировать следующим образом: `cut` запрещает использовать все утверждения определения, лежащие ниже, а также все альтернативные выводы конъюнкции целей, находящиеся левее `cut` в утверждении; `cut` не влияет на выполнение целей, находящихся правее в утверждении. На этом участке может быть построено несколько решений и, естественно, допустим возврат.

В языке Пролог используется ограниченная форма логического отрицания — отрицание по невыполнимости. Это понятие базируется на гипотезе о замкнутости мира. Суть ее сводится к предположению об истинности отрицания некоторой цели, если не существует возможности доказательства самой цели. Отрицание по невыполнимости достаточно просто определить с помощью уже введенных

средств языка. Для этого достаточно воспользоваться предикатами cut и fail:

not(G):— G, !, fail.  
not(G).

Как правило, большинство реализаций Пролога содержат предикат not в качестве системного.

Полезным расширением Пролога являются *металогические предикаты*, которые работают не с традиционными объектами логики первого порядка, а скорее воздействуют на процесс доказательства (выполнения программ) и на структуру программы. Примером их может служить предикат cut. К этому классу относятся и предикаты var, popvar, call, compound, integer, atom и constant. В Прологе предусмотрены и другие встроенные предикаты, в частности предикаты, предназначенные для модификации программы в процессе ее выполнения.

**BAF-метод** (Backtrack After Fail — возврат после отказа). Суть его заключается в организации повторяющихся отказов некоторой цели и возвратов, в результате выполнения целей, лежащих левее цели, приводящей к отказу, будет возобновляться. Естественно, наиболее подходящим кандидатом постоянно отказывающейся цели является fail. Проиллюстрируем использование BAF-метода на примере Пролог-программы для распечатки всех книг указанного автора, имеющихся в библиотеке:

печатать\_книги(X):— книга(X,Y), nl, write(Y), fail

База данных программы должна содержать информацию о книгах и авторах:

книга(толстой, война\_и\_мир).  
книга(толстой, аина\_кареннина).  
книга(достоевский, идиот),  
книга(достоевский, братья\_карамазовы).  
.....

С помощью этой же программы можно вывести и названия всех книг, имеющихся в библиотеке:

? — печатать\_книги(X).

Хотя выполнение запросов к программе печати книг будет всегда приводить к неуспешному завершению (отказу), в качестве побочного эффекта будут получены необходимые результаты.

**SAF-метод** (Cut And Fail — отсечение и отказ). В некоторых случаях при выполнении программы полезно ограничить поиск по базе данных программы. Повторяющееся выполнение целей можно организовать с помощью предиката fail. Естественным расширением этого приема является не сканирование всей базы данных, а организация «забывания» не просмотренной еще ее части при выполнении некоторого условия. Таким образом, для организации повторений можно, как и раньше, использовать возврат, а с помощью cut запретить его при выполнении определенных условий.

**UDR-метод** (User-Defined Repeat — повторения, управляемые пользователем). В отличие от предыдущих двух методов, в которых количество повторений ограничено общим количеством неопробованных альтернатив, здесь повторения могут выполняться произвольное число раз. Для этой цели во многих версиях Пролога используется встроенный предикат repeat:

repeat.  
repeat :— repeat.

В языке Пролог одним из основных методов управления выполнением программ является *рекурсия*. Как правило, Пролог-программа представляет собой совокупность рекурсивных или взаимно рекурсивных определений. Рекурсивное правило можно представить с помощью следующей схемы:

<Рекурсивное правило>:—  
<Список предикатов>.

<Предикат, определяющий условие выхода>,  
<Список предикатов>,  
<Рекурсивное правило>,  
<Список предикатов>.

В общем случае тело этого правила содержит пять групп: 1) предикаты, успешное или безуспешное выполнение которых не приводит к рекурсии; 2) предикат, определяющий условие выхода, успешное выполнение которого вызывает рекурсию; 3) отказ можно рассматривать как прекращение рекурсии; 4) непосредственно рекурсивное обращение; 5) предикаты, выполнение которых не влияет на рекурсию, в процессе рекурсивного выполнения они «выталкиваются» в стек и выполняются лишь по завершении рекурсии. Следует обратить внимание на условие завершения рекурсии. Рекурсивное правило должно всегда содержать этот предикат. Предложения Пролога, построенные по рассмотренной схеме, принято называть GRR-правилами (General Recursive Rule — общее правило рекурсии).

**GAT-метод** (Generate And Test — генерация и проверка). Этот метод состоит в том, что одна цель генерирует решения, а другая проверяет их допустимость. Если построенное решение не подходит, выполняется возврат к цели-генератору для построения альтернативы. Как правило, подобные определения концептуально более ясны, чем определения, построенные по другим принципам. Однако при этом за простоту часто приходится расплачиваться эффективностью.

Элегантность и наглядность рекурсивного программирования — часто слишком высокая плата за эффективность. Поэтому многие Пролог-системы способны самостоятельно трансформировать некоторые рекурсивные определения в итеративные. Такого рода преобразования могут выполняться для рекурсивных определений, называемых определениями с хвостовой рекурсией. После оптимизации определения с хвостовой рекурсией могут выполняться столь же эффективно, как итеративные. Однако не все случаи хвостовой рекурсии могут быть обнаружены Пролог-системой. В интересах программиста иногда явно указать такие определения с помощью метапредиката `cut`.

Пролог допускает программирование модификации программ с помощью специальных встроенных предикатов `clause`, `assert`, `assertz`, `retract`. Предикат `clause(H, B)` вызывается, если `H` — основной терм, в противном случае возбуждается состояние «Ошибка». Выполнение `clause(H, B)` заключается в выборе первого Пролог-предложения, голова которого унифицируется с `H`, и унификация `B` с телом выбранного предложения. При возврате опробуется следующее предложение и т. д. Согласно семантике этого предиката предложение может быть доступно только через голову. Предикат `assert(C)` добавляет в начало соответствующего определения новое предложение `C`, `assertz(C)` выполняет вставку предложения в конец соответствующего определения. Удаление предложений из определений можно реализовать с помощью предиката `retract(C)`. Для удаления, например, предложения `a`:—`a`. достаточно выполнить `retract((a:—C))`. Рассмотренные средства обеспечивают возможность программирования с побочными эффектами. Так, с их помощью легко организовать глобальные переключатели (там, где они действительно необходимы), сократить непомерно разросшийся список аргументов, выполнить модификацию программы. Тем не менее «создание программ с побочными эффектами часто следствие интеллектуальной лености и некомпетентности» [Sterling et al., 1986].

Более подробное описание языка и алгоритмов интерпретации можно найти в [Clocksin, 1981; Lloyd, 1984; van Emden, 1984; Клоксин и др., 1987]. Особое внимание следует обратить на великолепную книгу Шапиро и Стерлинга [Sterling et al., 1986].

## Реализация языка Пролог

Со времени появления первого Пролог-интерпретатора (1972 г.) сложился ряд методов, позволяющих достаточно эффективно реализовать обрабатываемый Пролог-системами алгоритм управления (семантическая линейная резолюция для хорновских дизъюнктов). На смену первым Пролог-системам интерпретирующего

типа постепенно стали приходиться более совершенные реализации, в основу которых положена идея компиляции Пролог-программ [Clocksin, 1981]. Исходная Пролог-программа компилируется в код некоторой абстрактной (например, стековой) машины, затем этот код может быть реализован программно, микропрограммно или аппаратно.

В настоящее время существует ряд [Warren, 1977; Bowen, 1983; Colmeaguer, 1983; Kotoowski, 1983 и др.] *абстрактных Пролог-машин*, предназначенных для самых разных целей. Одной из простейших, например, является «машина времени» Колмеагера, разработанная специально для иллюстрации особенностей выполнения Пролог-программ [Colmeaguer, 1983] (реализация возвратов и унификации). Среди наиболее удачных можно выделить абстрактную последовательную машину Уоррена [Warren, 1977], идеи которой были реализованы в мощной Пролог-системе DEC 10/20 компилирующего типа. Это, по-видимому, первая компилирующая Пролог-система. Позже последовательная машина Уоррена претерпела заметные изменения [Warren, 1983] и сейчас это наиболее удачная разработка, пригодная как для аппаратной, так и для программной реализации. Машина Уоррена часто рассматривается в качестве отправной при разработке не только чисто последовательных, но и конвейерных, а также некоторых параллельных систем [Tick et al., 1984; Nakazaki et al., 1985; Dobry et al., 1984; Nishikama, 1983; Dobry et al., 1985].

### Организация Пролог-машины

Систему команд абстрактной Пролог-машины (далее просто Пролог-машины) составляют стекоориентированные команды высокого уровня. В состав машины включены различные стеки и регистры, в которых запоминаются состояния Пролог-машины (вычислительного процесса), соответствующие точкам ветвления программы. Возвраты реализуются как восстановления состояния Пролог-машины. Эффективность унификации достигается в основном за счет теговой организации памяти и отказа от проверки циклических вхождений переменных в термы при унификации (occurency check). Здесь в отличие от интерпретирующих Пролог-систем непосредственно выполнению программы предшествует этап компиляции в исполняемый код Пролог-машины. Компиляция возможна из-за использования в Прологе некоторой формы входной резолюции. Благодаря этому при резолюции всегда один из двух хорновских дизъюнктов принадлежит к известному заранее множеству входных дизъюнктов, определяемому исходной Пролог-программой.

Однородная память Пролог-машины разделяется на несколько областей, предназначенных для хранения различных объектов Пролог-программы: область кода программы; куча (стек глобальных); стек (стек локальных); стек восстановления; стек унификации (PDL) (рис. 1.5). Область кода программы остается неизменной в процессе ее выполнения (особенности реализации встроенных предикатов типа assert здесь не рассматриваются). Остальные области программы (стеки), как правило, динамически изменяются: по мере выполнения целей в них «вталкивается» информация и «вытаскивается» при возвратах, отработке хвостовой рекурсии [Warren, 1980] и выполнении cut-операторов [Clocksin et al., 1981].

Объекты Пролога представляются в виде 32-разрядных слов, снабженных тегами длиной 2 бит или более (например, 8+32 в персональной Пролог-машине PSI ICOT [Nishikama, 1983] или 4+32 в РЕК Kobe University [Kaneda et al., 1986]).

Для представления сложных объектов в памяти Пролог-машины, таких как термы-структуры (списки и структуры) и Пролог-предложения, наиболее часто применяются методы копирования и разделения (некопирования) структур. В Пролог-машине Уоррена используются оба метода: термы сложной структуры представляются посредством копирования, а цели, образующие резолюенту, — разделения структур. При *копировании структур* описание структуры разбивается на две части и состоит из скелета структуры и заполнения структуры. Первая часть — скелет структуры — неизменна в процессе выполнения Пролог-программы и хранится, как правило, в единственном экземпляре. При каждом использовании

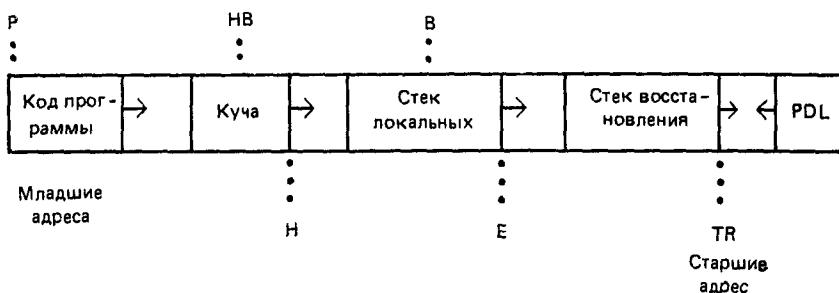


Рис. 1.5. Распределение памяти в Пролог-машине

структуры создается только собственный экземпляр заполнения. Таким образом, скелет структуры хранится отдельно и не дублируется при многократном использовании структуры. Характерной особенностью метода копирования структур является возможность организации быстрого доступа к отдельным компонентам структуры, хранимым в области заполнения, и не очень эффективное использование памяти. Очевидно, что источник неэффективного использования памяти в дублировании многократно используемых в структуре переменных. Причем чем больше таких переменных, тем больше область заполнения и тем больше данных копируется.

Переменные, входящие в Пролог-предложение, локальны для этого предложения. Естественно, локальны для предложения и переменные, входящие в структуру. В методе *разделения структур* [Wagner, 1982] при каждом использовании предложения расположение переменных в его среде неизменно. Под средой обычно понимается последовательность (вектор) переменных, входящих в предложение. Среда предложения создается при каждом его выполнении. При предположении о неизменности относительного порядка размещения переменных в среде предложения структура задается в виде трех компонентов: скелет структуры; указатель на среду Пролог-предложения; таблица относительных адресов (порядковых номеров или смещений относительно начала среды), определяющих положение компонентов структуры в среде.

Метод разделения структур предпочтительнее, если в программе многократно используется незначительное число одних и тех же сложных структур; если обрабатывается много простых структур, предпочтительнее метод копирования. В среднем по затратам памяти разделение структур несколько лучше, чем копирование [Mellish, 1981].

**Стеки Пролог-машин.** Под каждый объект (переменную, константу, структуру или список) отводится по одному слову. Тип объекта хранится в теге слова [Wagner, 1983]. Такое расположение позволяет использовать простую безопасную стратегию связывания переменных переменными: связывание переменной с большим адресом переменной с меньшим адресом. Любая переменная стека глобальных (кучи) имеет меньший адрес, чем адрес переменной стека локальных, что необходимо для исключения появления опасных ссылок при удалении или сокращении среды в стеке локальных.

Область кода содержит исполняемый код программы, получаемый в результате компиляции Пролог-программы. Куча предназначена для хранения составных объектов данных (структуры и списки), а также переменных, глобализуемых при выполнении программы. Куча работает как стек с дисциплиной обслуживания LIFO. Допускается и интенсивно используется адресный доступ к элементам кучи. Это — единственная область памяти, для которой может потребоваться выполнение сборки мусора.

В стеке локальных могут храниться элементы двух типов: среды предложения и точки ветвления. Помимо вектора переменных тела некоторого предложения

среда содержит продолжение. Продолжение представляет собой указатель в область кода программы и содержит адреса следующей выполняемой цели *родительского* Пролог-предложения и его среды. Точка ветвления предназначена для хранения всей необходимой информации для восстановления состояния Пролог-машины в случае отказа (неуспешного выполнения) некоторой цели. Состояние Пролог-машины имеет смысл восстанавливать только там, где существуют альтернативы выбора Пролог-предложений. Поэтому точки ветвления создаются не для каждой процедуры, а лишь там, где опробованы еще не все Пролог-предложения. При создании в каждой точке ветвления запоминаются указатели на альтернативное Пролог-предложение (ALT), голову кучи (H), продолжение программы (CP), последнюю точку ветвления (B), среду в стеке (E), голову стека восстановления (TR) и количество аргументов вызываемой процедуры (N), а также регистры аргументов (AI — Am, где m — число аргументов процедуры).

Кроме запоминаемых в точке ветвления регистров и указателей в состав Пролог-машины входят: P — счетчик команд (указатель в область кода программы), A — указатель на верхушку стека, HB — указатель на адрес в куче, соответствующий точке возврата (адрес верхушки кучи, соответствующей B); S — указатель на (существующую) структуру в куче; X1, X2, ... — регистры временных переменных.

Стек восстановления (trail) содержит указатели на переменные — адреса переменных — в стеке и в куче. Адреса этих переменных заносятся в стек восстановления при связывании в процессе унификации и «выталкиваются» при возврате. При «выталкивании» соответствующие переменные приобретают статус свободных. Дисциплина работы стека восстановления — LIFO.

Существует еще один небольшой стек, предназначенный для выполнения унификации вложенных структур и списков PDL. Назначение этого стека будет рассмотрено позже.

Пример состояния стеков после вызова процедуры, представленной несколькими альтернативными Пролог-предложениями, приведен на рис. 1.6.

**Типы переменных.** Все переменные Пролог-предложения подразделяются на два типа — постоянные и временные.

К *постоянным* относятся переменные, значения которых могут потребоваться на последующих шагах вывода, в командах W-кода они обозначаются через Yi (Yi — номер переменной, интерпретируемый как смещение относительно начала среды). Такие переменные хранятся в среде, размещаемой в стеке.

К *временным* относятся переменные, значения которых актуальны ограниченный промежуток времени (только до вызова очередной цели Пролог-предложения), либо переменные, входящие в состав структур и размещенные в куче в качестве их аргументов. Временные переменные располагаются в регистрах и обозначаются Xi.

Правило определения типа переменной следующее [Wagren, 1983]. Временной является переменная, имеющая первое вхождение либо в голову Пролог-предложения, либо в структуру (список), либо в последнюю цель Пролог-предложения и при этом входящая не более чем в одну цель тела. Голова Пролог-предложения при подсчете количества вхождений рассматривается как часть первой цели. Все остальные переменные трактуются как постоянные. Предложения, содержащие менее двух целей в теле, и факты не имеют постоянных переменных. Приведенное правило верно лишь для копирования структур и не работает при разделении структур.

В ряде случаев перед выполнением последней цели среда Пролог-предложения может быть удалена из стека локальных (оптимизация хвостовой рекурсии). Если значения некоторых постоянных переменных становятся ненужными для выполнения последующих целей, они удаляются из среды. Это возможно благодаря порядку размещения постоянных переменных в среде. Сокращение среды Пролог-предложения, аналогичное оптимизации хвостовой рекурсии, возможно лишь в случае, если она создана позже последней точки возврата.

Типы переменных Пролог-предложения и порядок их расположения в среде определяются на этапе трансляции Пролог-программы в исполняемый код. По-

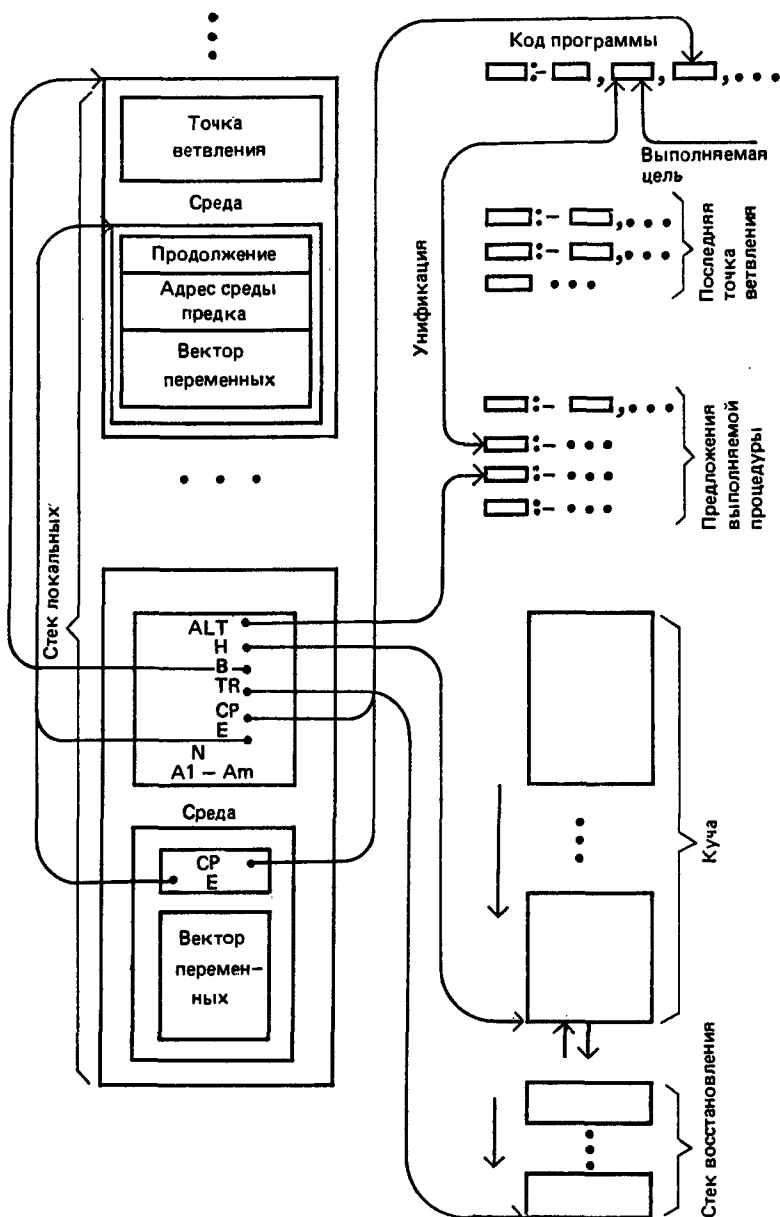


Рис. 1.6. Стек и регистры при вызове процедуры, представленной несколькими альтернативными предложениями



Таблица 1.1

## Команды W-кода

Команды чтения	get_variable get_value get_nil get_constant get_list get_structure	X <sub>i</sub> /Y <sub>i</sub> , A <sub>j</sub> X <sub>i</sub> /Y <sub>i</sub> , A <sub>j</sub> A <sub>j</sub> с, A <sub>j</sub> A <sub>j</sub> f, A <sub>j</sub>	X <sub>i</sub> — временная переменная с номером i; Y <sub>i</sub> — постоянная переменная с номером i; A <sub>j</sub> — регистр аргумента с номером j; с — константа; f — главный функтор структуры
Команды записи	put_variable put_value put_unsafe_value put_nil put_constant put_list put_structure	X <sub>i</sub> /Y <sub>i</sub> , A <sub>j</sub> X <sub>i</sub> /Y <sub>i</sub> , A <sub>j</sub> Y <sub>i</sub> , A <sub>j</sub> A <sub>j</sub> с, A <sub>j</sub> A <sub>j</sub> f, A <sub>j</sub>	
Команды унификации	unify_variable unify_value unify_unsafe_value unify_nil unify_constant unify_cdr	X <sub>i</sub> /Y <sub>i</sub> X <sub>i</sub> /Y <sub>i</sub> X <sub>i</sub> /Y <sub>i</sub> с	
Команды работы с процедурами	call execute proceed allocate deallocate	mp/p, m mp/p	
Команды индексирования и организации цепочек	try retry trust try_me_else retry_me_else trust_me_else switch-on_term switch-on_constant switch-on_structure	L L L L L fail (Cr, Cc, Cl, Cs) R, table R, table	L, Cr, Cc, Cl, Cs — метки в коде программы; table — хэш-таблица R — размер хэш-таблицы

стоянную переименовую можно удалить из среды непосредственно перед выполнением цели Пролог-предложения, в которую она имеет последнее (слева направо) вхождение. Переменные, остающиеся в среде после ее сокращения, называются актуальными на данный момент выполнения программы.

**Система команд (W-код).** Команды W-кода подразделяются на пять групп: команды чтения, записи, унификации, работы с процедурами и команды индексирования и организации цепочек (табл. 1.1).

**Команды чтения** выполняют действия, связанные с унификацией аргументов текущей цели и головы Пролог-предложения. Информация об аргументах цели передается через регистры A<sub>j</sub> (предполагается, что ссылка на значение первого аргумента записана в регистре A<sub>1</sub>, второго — в A<sub>2</sub> и т. д.). В зависимости от

типа аргумента головы Пролог-предложения используется та или иная команда чтения. Команда `get_constant c, Aj` соответствует аргументу-константе. Для константы `nil` введена специальная команда `get_nil`. Унификации с аргументом-переменной соответствуют команды `get_variable` и `get_value`. Команда `get_variable Xi` (или `Yi`), `Aj` указывается для первого вхождения переменной в голову Пролог-предложения, она заведомо является несвязанной, и содержимое регистра `Aj` просто переписывается в соответствующую переменной ячейку памяти. Во всех остальных случаях используется команда `get_value`. Здесь переменная может уже оказаться связанной, и необходима проверка сопоставимости ее значения со значением аргумента. После необходимого этапа замыкания ссылок проверяются теги значений аргументов. При унификации, например, двух несвязанных переменных переменная с большим адресом связывается переменной с меньшим адресом. Так возникают цепочки ссылок, которые могут быть достаточно длинными. Для поиска действительного значения переменной необходимо пройти по всей цепочке и найти ее последний элемент, который может быть либо термом, либо несвязанной переменной. Операция прохождения по цепочке называется замыканием ссылок или переадресацией. В зависимости от реализации во время выполнения этой операции цепочка может разрушаться. В этом случае во все ее элементы записывается ссылка непосредственно на найденное значение, т. е. на последний элемент цепочки. Несовпадение тегов аргументов сигнализирует о невозможности унификации — константа несопоставима со структурой, структура со списком и т. д. Если какому-либо аргументу соответствует несвязанная переменная, она связывается значением другого аргумента. Совпадение тегов — один из признаков возможности унификации. Если унификация двух констант достаточно очевидна, для унификации структур и списков в общем случае используется дополнительный стек PDL. Структуры сопоставляются поаргументно, в порядке обхода вложенных структур «сначала в глубину». В стек PDL «вталкиваются» указатели на следующие аргументы структур, которые не были еще унифицированы. Глубина стека PDL при таком порядке унификации равна глубине вложенности структур. Возможно также использование порядка обхода вложенных структур и списков «сначала в ширину». Здесь максимальная глубина стека будет равна количеству аргументов структуры. Однако, как правило, в Прологе структуры имеют тенденцию быть «шире», а не «глубже», и поэтому использование обхода «сначала в глубину» предпочтительнее [Fagin et al., 1985].

Команды `get_structure` и `get_list` используются для унификации аргумента цели с явно заданной в голове Пролог-предложения структурой и списком соответственно. В отличие от команды `get_value` для команд `get_structure` и `get_list` можно явно задать последовательность команд унификации, передавая указатели на аргументы структур (списков), в свою очередь являющихся сложными термами, через регистры `Xi`.

Команды записи предназначены для формирования значений аргументов цели Пролог-предложения. В регистр `Aj` помещается ссылка на объект, указанный на месте первого аргумента соответствующей команды записи W-кода. При необходимости выполняется замыкание ссылок. Для каждой команды записи можно найти аналогичную ей команду чтения. Однако для команды `put_unsafe_value Yi, Aj` такого аналога нет, она соответствует последнему вхождению в тело Пролог-предложения «ненадежной» переменной, т. е. постоянной переменной, возможно оставшейся неопределенной к моменту выполнения цели, в которую она имеет последнее вхождение. В этом случае в регистр `Aj` помещать адрес переменной в стеке локальных нельзя, так как после сокращения среды в регистре `Aj` может образоваться опасная («висячая») ссылка. С другой стороны, не всегда разумно помещать признак несвязанной переменной в регистр `Aj`, так как это потребует введения достаточно громоздкого механизма отслеживания ссылок, связывающих регистры `Aj` и `Xi`; в некоторых случаях такие связи могут приводить к нежелательным последствиям. Поэтому возникает необходимость в глобализации несвязанной ненадежной переменной. В куче выделяется ячейка памяти, а в регистре `Aj` указывается ее адрес. Глобализация нужна и в том случае, когда ненадежная переменная ссылается на несвязанную переменную, расположенную

в текущей среде. Здесь глобализуемой является именно эта несвязанная переменная. Аналогичная проблема возникает при появлении команды `put_variable Xi, Aj`, по которой происходит безусловная глобализация временной переменной `Xi`. Команды `put_structure` и `put_list` предназначены для размещения в куче и подготовки к унификации структуры или списка, указанных на месте аргумента некоторой цели Пролог-предложения.

Команды унификации используются в случае, когда структура или список явно заданы на месте аргумента головы или цели Пролог-предложения, и предназначены как для унификации таких структур с ранее созданными структурами, так и для построения новых структур в куче. Последовательности команд этой группы обязательно предшествует команда `get` или `put` для структуры или списка, которая устанавливает режим выполнения следующих за ней команд унификации. Режим записи соответствует построению новой структуры (списка) в куче, а режим чтения — унификации с уже существующей структурой (списком). Команды `put` всегда определяют режим записи, а команды `get` — режим чтения, если в регистре `Aj` содержится ссылка на структуру (список), в противном случае устанавливается режим записи. В режиме чтения команды группы выполняют унификацию аргументов структуры, указанной в Пролог-предложении, с аргументами структуры, адресуемой через регистр `S` (указатель на существующую структуру); загрузка регистра `S` осуществляется командой `get` для структуры или списка. В режиме записи конструируется последовательность аргументов новой структуры, адресуемой через регистр `H` (указатель на вершину кучи).

Последовательность команд унификации определяется способом хранения сложных термов в куче. Для метода копирования структур можно выделить два основных способа кодирования — ориентированного на структуру и списки. И первому, и второму способу присущ ряд недостатков. Например, для структуро-ориентированного способа характерно увеличение последовательности команд унификации при работе со списками, а для спискоориентированного заметно увеличиваются затраты памяти, необходимые для хранения структур в куче. Поэтому на практике часто используется еще один способ, представляющий собой комбинацию обоих способов [Dobry et al., 1984]. Так же как и в структуро-ориентированном способе, под аргументы структуры здесь выделяется непрерывный участок памяти кучи. Пока существует такая возможность, элементы списка располагаются в последовательных ячейках памяти. При необходимости часть элементов списка может быть перенесена в другой участок памяти. Этот способ сочетает в себе достоинства обоих предыдущих. К его недостаткам можно отнести некоторое усложнение семантики команд унификации и необходимость введения специального бита, сигнализирующего, что данная ячейка памяти является ссылкой на продолжение списка, а не его очередным элементом.

Команды `unify_value` и `unify_unsafe_value` используются, если для соответствующей переменной уже была задана команда `get_variable`, `put_variable` или `unify_variable`. Выполнение их заключается в унификации очередного аргумента структуры с переменной, указанной на месте аргумента команды `W`-кода. В режиме записи под аргумент структуры предварительно отводится ячейка памяти в куче. Если заранее известно, что значением переменной является объект кучи (или константа), предпочтительно использовать команду `unify_value`, которая просто запишет ссылку на этот объект в отведенную ячейку. В противном случае необходимо использовать команду `unify_unsafe_value`, которая после замыкания ссылок определяет, связывать аргумент переменной или, наоборот, переменную аргументом. В списке команд унификации нет команд `unify_structure` и `unify_list`, так как вместо них используются команды `get_structure` и `get_list` соответственно. В некоторых реализациях для работы с хвостом списка вводится специальная команда унификации `unify_cdr`.

На команды работы с процедурами возложены функции управления выполнением программы и выделения памяти под среду вызываемой процедуры. Команды `call`, `execute` и `proceed` предназначены для организации управления. Первые две команды передают управление процедуре, унифицирующейся с соответствующей целью Пролог-предложения. В команде `call` указывается также количество

(актуальных) постоянных переменных  $m$  среды текущего Пролог-предложения, значения которых могут потребоваться на последующих шагах вывода. При отсутствии точек ветвления ниже элемента стека локальных, соответствующего выполняемому Пролог-предложению, следующий элемент будет размещаться в стеке, частично перекрывая среду последнего элемента стека. Такое «поджатие» стека позволяет сократить расходы памяти, необходимой для выполнения Пролог-программы. Команда `execise` заменяет команду `call` при вызове последней цели Пролог-предложения. Количество актуальных постоянных переменных здесь не указывается, так как оно, очевидно, равно нулю. Оптимизация хвостовой рекурсии может выполняться перед командой `execise`. Команда `proceed` осуществляет возврат на продолжение программы в случае успешного выполнения факта. Продолжение программы определяется содержимым регистров `CP` и `E`. Пара команд `allocate` и `deallocate` предназначена для создания и удаления соответственно среды Пролог-предложения. (Напомним, что среда Пролог-предложения создается, только если тело Пролог-предложения содержит не менее двух целей.) По существу, команды работы с процедурами манипулируют содержимым регистров `CP` и `E`, а при передаче управления загружают в регистр `P` адрес следующей выполняемой процедуры.

В простейших реализациях Пролога при возвратах последовательно опробываются все альтернативные предложения процедуры. Однако, если на месте  $i$ -го аргумента в голове предложения записана, например, структура, а в цели — список, ясно, что унификация цели с Пролог-предложением закончится отказом. Анализируя заранее предложения процедуры, можно построить цепочки, состоящие из потенциально подходящих предложений для того или иного типа аргумента цели. В Пролог-машине Уоррена предлагается рассматривать в качестве ключа для функции выделения (индексирования) первый аргумент Пролог-предложения. Для связывания в цепочки потенциально сопоставимых с заданной целью Пролог-предложений служат команды *индексирования* и *организации цепочек*. Первой командой процедуры, предложения которой будут индексироваться, является команда переключения по типу первого аргумента цели, т. е. по тегу регистра `A1`, `switch_on_term` (`Cr`, `Cc`, `Cl`, `Cs`). Каждому типу аргумента — переменной, константе, списку или структуре — соответствует своя цепочка Пролог-предложений; `Cr`, `Cc`, `Cl` и `Cs` — адреса начала цепочек, соответствующих перечисленным типам аргументов. Очевидно, что с переменной может быть унифицирован любой объект, поэтому для цели с переменной на месте первого аргумента должна быть построена цепочка, состоящая из всех предложений процедуры. Аналогично для цели с первым аргументом-списком формируется цепочка из предложений, содержащих на месте первого аргумента головы переменной или список. В отличие от этого для констант и структур предлагается использовать двойную индексацию — сначала по типу аргумента, а затем по имени константы (или фактора для структур). Для каждой Пролог-процедуры строятся хэш-таблицы констант и факторов. По команде `switch_on_term` управление передается по адресу `Cc` (или `Cs`) где записана команда `switch_on_constant` `R`, `table` (или `switch_on_structure` `R`, `table`), где `R` — размер хэш-таблицы. Выполнение команды `switch_on_constant` или `switch_on_structure` заключается в поиске элемента хэш-таблицы, соответствующего входной константе (записанной в литерале цели) или фактору входной структуры и определению адреса начала цепочки, соединяющей предложения, содержащие одноименные константы или структуры на месте первого аргумента головы. При необходимости в эти цепочки включаются и предложения с переменной на месте первого аргумента головы. Конечно, для констант или факторов можно организовать одну общую цепочку, не прибегая к хэш-таблице. Однако увеличение длины цепочки приведет к существенному снижению эффективности индексирования по этим типам объектов.

Для организации цепочек из предложений Пролог-процедуры используются команды `try-me-else L`, `retry-me-else L`, `trust-me-else fail`, `try L`, `retry L`, `trust L`, где `L` — некоторая метка в `W`-коде программы. Семантика первых трех команд несколько отличается от семантики последних трех. По командам `try-me-else L`, `retry-me-else L`, `trust-me-else fail` управление передается следующей команде

W-кода, а при отказе—команде с меткой L (отказ для данной цепочки Пролог-предложений формируется по команде `trust_me_else`). По командам `try L`, `retry L`, `trust L` управление сначала передается команде W-кода по метке L и при отказе—на следующую команду W-кода (отказ для данной цепочки предложений формируется по команде `trust`). Команды `try_me_else` и `try` определяют начало цепочки предложений и сигнализируют о наличии альтернативы. Для цепочки, состоящей из одного элемента, команды организации цепочек не используются. По командам `try_me_else` и `try` в стеке создается точка ветвления. Команды `retry_me_else` и `retry` описывают не первый и не последний элемент цепочки, а изменяют лишь адрес альтернативного предложения в точке ветвления. Команды `trust_me_else` и `trust` соответствуют последнему элементу цепочки, предназначены для удаления созданной командой `try_me_else` или `try` точки ветвления. Как видно из семантики команд `try`, `retry` и `trust`, они должны располагаться в W-коде программы непосредственно друг за другом, в то время как команды `try_me_else`, `retry_me_else` и `trust_me_else` указываются перед тем Пролог-предложением, альтернативу для которого они описывают.

Для решения реальных задач искусственного интеллекта необходимы машины, скорость которых должна превышать скорость света, а это возможно лишь в параллельных системах. Поэтому последовательные реализации следует рассматривать как рабочие станции для создания программного обеспечения будущих высокопроизводительных параллельных систем, способных выполнять сотни миллионов логических выводов в секунду. В настоящее время существуют десятки моделей параллельного выполнения логических программ вообще и Пролог-программ в частности. Часто это модели, использующие традиционный подход к организации параллельных вычислений: множество параллельно работающих и взаимодействующих друг с другом процессов. В последнее время значительное внимание уделяется и более современным схемам организации параллельных вычислений—потокowym моделям [Wise, 1986; Goto et al., 1984; Conery et al., 1981]. В моделях параллельного выполнения рассматривается традиционный Пролог и присущие ему источники параллельности.

Появились новые языки логического программирования *Concurrent Prolog* [Shapiro et al., 1983], *Parlog* [Clark et al., 1984], *Epilog* [Wise, 1986], *GHC* [Тапакэ et al., 1986] и др., специально ориентированные на параллельное выполнение программ. Естественно, что потребовалось изменение исходной семантики Пролога. Однако так или иначе во многих моделях параллельного выполнения эксплуатируется два источника параллельности, присущие и традиционному Прологу: И- и ИЛИ-параллельность. Под И-параллельностью понимается параллельное выполнение целей тела Пролог-предложения. ИЛИ-параллельность предполагает параллельное выполнение Пролог-предложений, потенциально сопоставимых с выполняемой целью. Одной из первых работ по исследованию ИЛИ-параллельности была работа Конери [Conery et al., 1981], в которой рассматривается выполнение Пролог-программ в терминах И/ИЛИ-процессов. Но среди множества логических языков параллельного программирования в последние годы лидирует, пожалуй, *Concurrent Prolog*. Программа здесь определяется как совокупность охраняемых предложений вида «A:—охрана|тело.» и трактуется в терминах процессов следующим образом: процесс A сводится к системе процессов, представляемой частью «тело» предложения. Каждая цель правой части предложения интерпретируется как процесс, а разделяемые целями переменные—как каналы связи между процессами. Процесс A может замещаться системой процессов и завершается, когда он сводится к пустой системе. Для эффективного управления передачей сообщений от процесса к процессу в *Concurrent Prolog* используются аннотированные переменные. Для сведения процессов могут пробоваться несколько альтернативных предложений его определения. Операция «снятия охраны» (|), разделяющая компонент охраны и тело предложения, ограничивает ИЛИ-параллельность при сведении процессов. Основной формой параллелизма в *Concurrent Prolog* является И-параллельность при ограниченной ИЛИ-параллельности. Однако тотальная И-параллельность может быть ограни-

цена за счет использования аннотированных переменных. По сути аннотированные переменные служат для организации синхронизации выполнения И-параллельных процессов.

При выборе языка Пролог как базового языка программирования в японском проекте вычислительных систем пятого поколения в качестве одного из его недостатков отмечалось отсутствие развитой среды программирования и непригодность Пролога для создания больших программных систем. Сейчас ситуация несколько изменилась, хотя говорить о действительно ориентированной на логическое программирование среде преждевременно. В последнее время появились языки логического программирования с развитыми средствами модульности (например Himiko, ESP, Shape-Up и др.). Существуют работы и по Пролог-ориентированным системам управления программными проектами (ELOG), системам графической отладки, верификации и синтеза логических программ (ARGUS/5, PROEDIT) [Kanamori et al., 1986]. Интересны работы по графическому отладчику для TRM (Transporant Prolog Machine) и другие исследования в этой области.

## 1.5. Метаалгоритмический язык Рефал и тенденции его развития

*С. А. Романенко*

Язык Рефал — алгоритмический язык рекурсивных функций — был создан в качестве абстрактного метаязыка, предназначенного для описания различных, в том числе и алгоритмических, языков [Турчин, 1966; Турчин, 1968; Базисный, 1977] и различных видов обработки таких языков. При этом, в частности, имелось в виду и использование Рефала в качестве метаязыка над самими собой. Рефал был задуман как метаалгоритмический язык, но для пользователя это язык обработки символической информации. Поэтому, помимо описания семантики алгоритмических языков, он нашел и другие не менее важные применения. В первую очередь, это выполнение громоздких аналитических выкладок в теоретической физике и прикладной математике, интерпретация и компиляция языков программирования, доказательство теорем, моделирование целенаправленного поведения, а в последнее время и задачи искусственного интеллекта. Общим для всех этих применений являются сложные преобразования над объектами, определенными в некоторых формализованных языках.

### Основные понятия

Из специфики Рефала как метаязыка вытекает специфика его основных объектов: это должны быть языковые объекты, общие для самых разных языков. Такими объектами можно считать последовательности символов, структурированные в виде дерева.

Элементарные единицы для построения объектов Рефала делятся на два вида: специальные знаки и объектные символы. В каждой конкретной реализации Рефала множество объектных символов (знаков) — это тот набор литер, который обеспечен устройствами подготовки и отображения данных (перфораторами, дисплеями, принтерами). Из знаков строятся более крупные единицы — символы, термы и выражения.

*Символ* — минимальная семантическая единица, не расчленимая на части средствами языка Рефал. Среди символов выделяются составные символы вида /тело\_составного\_символа/, где тело\_составного\_символа — последовательность литер, не содержащая литеры «/». Множество составных символов в зависимости от вида их тела разбивается на непересекающиеся классы: символы-метки, символы-числа и символы-ссылки. *Символами-метками* называются составные символы, телом которых является идентификатор, *символами-числами* — составные символы, телом которых представляет собой целое неотрицательное число,

*символами-ссылками* — составные символы, тело которых начинается с литеры «%», вслед за которой идет шесть 16-ричных цифр. Символы-метки используются в программах в качестве имен функций, символы-числа — для обозначения чисел. Назначение и использование символов-ссылок обсуждается ниже.

К специальным знакам, кроме уже упоминавшегося ограничителя составных символов, относятся *скобки: структурные* («(ки)»), *функциональные* (активационные). Последние называют левой и правой конкретизационными скобками и обозначают «K» и «.» соответственно (в некоторых реализациях их разрешается заменять «<» и «>»). Свободные переменные разных типов в простейшем случае обозначаются указателем типа переменной и ее индексом.

*Выражением* называется произвольная последовательность символов, переменных и скобок, правильно построенная относительно скобок, *термом* — выражение, которое представляет собой либо символ, либо выражение, заключенное в структурные или функциональные скобки. Таким образом, всякое выражение есть последовательность из некоторого (быть может, нулевого) числа термов.

### Рефал-программы и Рефал-функции

Программа, написанная на Рефале, определяет некоторый набор функций. Каждая из этих функций имеет один аргумент, значениями которого могут быть выражения, причем только такие, которые не содержат функциональных скобок и переменных. В Рефале вызов функции заключается в функциональные скобки, а имя функции указывается с помощью символа-метки: K/FUNC/[ARG]. Описание каждой функции имеет вид

$$\begin{aligned} \text{FUNC} \quad [L-1] &= [R-1] \\ [L-2] &= [R-2] \\ [L-N] &= [R-N] \end{aligned}$$

где FUNC — имя функции, а  $[L-i] = [R-i]$  — Рефал-предложения или правила конкретизации. Каждое предложение является указанием для замены одного выражения на другое, поэтому оно состоит из левой  $[L-i]$  (заменяемое выражение) и правой  $[R-i]$  (результат замены) частей. Синтаксически  $[L-i]$  и  $[R-i]$  являются выражениями.

Во многих случаях возникает необходимость из программ, написанных на Рефале, вызывать программы, написанные на других языках или непосредственно в командах ЭВМ. Эта необходимость возникает, в частности, когда требуется выполнять операции ввода-вывода, операций над числами и другие действия, которые Рефал-машина «не умеет» выполнять непосредственно или выполняет неэффективно. Функции, описанные не на Рефале, которые тем не менее можно вызывать обычным способом из программ, написанных на этом языке, называются *первичными* (часто такие функции называют машинными операциями). Собственно говоря, с точки зрения Рефала первичные функции — это просто некоторые функции, внешние по отношению к данной программе, поэтому, вызывая какую-либо функцию, можно даже не знать, что это — первичная функция или функция, написанная на Рефале. Разница состоит только в том, что исполнение вызова первичной функции занимает один шаг с точки зрения Рефал-машинны, а исполнение одной функции может занять несколько шагов.

Семантика Рефал-программы описывается в терминах абстрактной Рефал-машины. Рефал-машина имеет *поле памяти* и *поле зрения*. По сути дела, в поле памяти Рефал-машинны помещается программа, а в поле зрения — данные, которые будут обрабатываться с ее помощью, т. е. перед началом работы в поле памяти заносится описание набора функций, а в поле зрения — выражение, подлежащее обработке.

Пусть, например, в поле памяти находится единственное предложение

$$xxx = '137'$$

а в поле зрения — выражение

$$k/xxx/.$$

Тогда Рефал-машина заменит содержимое поля зрения на выражение '137' и остановится, поскольку в поле зрения не осталось ни одной пары функциональных скобок. Пусть в поле памяти два предложения

$$\begin{aligned} xxx &= '137' \\ &= '274' \end{aligned}$$

Тогда Рефал-машина просматривает предложения в том порядке, в котором они стоят в описании функции, и применяет первое из них, оказавшееся «подходящим», в данном случае «k/xxx/.» будет заменено на '137'.

Поле зрения может содержать сколько угодно много функциональных термов, которые могут быть как угодно вложены друг в друга. Некоторые из них являются самими внутренними, т. е. не содержат внутри себя других функциональных термов, самый левый из таких термов называется ведущим.

Работает Рефал-машина по шагам, в начале каждого шага ищется ведущий функциональный терм. Пусть это терм вида «k/FUNC/[ARG].», где FUNC — имя некоторой функции, а [ARG] — объектное выражение. Тогда Рефал-машина ищет описание функции FUNC и начинает сравнивать ARG с левыми частями предложений в описании этой функции. Пусть i — номер самого первого предложения, для которого [L—i]=[ARG]. Тогда Рефал-машина производит замену ведущего функционального термина, т. е. k/FUNC/[ARG] заменится на правую часть предложения [R—i], после чего переходит к исполнению следующего шага. Если же Рефал-машина не находит ни одного предложения, для которого [L—i]=[ARG], она сообщает, что «отождествление невозможно», и останавливается. Это означает, что либо программа на Рефале, либо исходные данные для нее заданы неверно. Работа Рефал-машины продолжается до тех пор, пока в поле зрения имеется хотя бы один функциональный терм. Если же в поле зрения после завершения очередного шага не остается ни одного функционального термина, Рефал-машина сообщает, что «вычисление окончено» и останавливается. При этом результатом работы Рефал-машины считается выражение, которое находится в поле зрения.

Данные средства позволяют описывать только функции, области определения которых — конечные множества, ибо для каждого конкретного значения аргумента приходилось предусматривать особое предложение. Ясно, что нужно уметь записывать предложения, применимые более чем к одному объектно-му выражению. Для этого нужно ввести в предложения переменные, которые при различных применениях предложения могут принимать разные значения.

### Переменные и спецификаторы

В Рефале используются переменные четырех типов: S-переменные, значения которых могут быть только символы; W-переменные, значения которых могут быть только термы; V-переменные, значения которых могут быть только непустые выражения; E-переменные, значениями которых могут быть выражения (в том числе и пустые). В общем случае любая переменная имеет следующий вид:

[Признак типа] [Спецификация] [Индекс]

где [Признак типа] — это один из специальных знаков «S», «W», «V» или «E», указывающих, к какому из четырех вышеперечисленных типов принадлежит переменная; [Спецификация] — описание дополнительных условий, налагаемых на множество допустимых значений переменной; [Индекс] — цифра либо буква русского или латинского алфавита, служит для того, чтобы различать между собой переменные.

Спецификация может иметь одну из следующих форм:

$$\left\{ \begin{array}{l} \text{[пусто]} \\ \text{[имя спецификатора]} \\ \text{([спецификатор])} \end{array} \right\}$$



Спецификация может быть пустой; в этом случае считается, что на возможные значения переменной не налагается никаких дополнительных ограничений. Например, значением «SX» может быть любой символ, значением «S('abc')X» могут быть только символы-литеры а, в или с. Если спецификация не пуста, она представляет собой либо имя спецификатора, либо непосредственно сам спецификатор, заключенный в скобки. Имя спецификатора, как и символ-метка, является идентификатором, только в качестве ограничителей используется не знак «/», а знак «:»:

: [идентификатор]:

Каждый спецификатор представляет собой описание некоторого множества термов, которое строится исходя из некоторого набора элементарных множеств. Обозначения этих элементарных множеств называются *элементами*. В качестве элементов могут использоваться:

1) имена других спецификаторов, в этом случае имя спецификатора обозначает множество, которое задает именуемый им спецификатор;

2) любой символ — множество, состоящее из одного символа [S], изображается самым этим символом;

3) конечное множество «стандартных» элементов:

S — множество всех символов,

В — множество термов вида «([E])», где [E] — произвольное объектное выражение,

W — множество всех термов,

F — множество символов-меток,

N — множество символов-чисел,

R — множество символов-ссылок,

O — множество символов-литер (объектных знаков),

L — множество букв (русских и латинских),

D — множество десятичных цифр.

Последовательность элементов спецификатора называется *цепочкой* элементов. Цепочка элементов вида  $x_1 x_2 \dots x_N$  обозначает множество  $x_1 + x_2 + \dots + x_N$ , которое представляет собой объединение множеств, соответствующих элементам цепочки. Например, «LD», обозначает множество букв и цифр.

В общем случае спецификатор имеет вид  $p_1(Q_1)p_2(Q_2) \dots p_N(Q_N)p_0$ , где «rk» и «Qk» — произвольные (могут быть пустые) цепочки элементов спецификатора. Ниже приведены примеры спецификаторов, для каждого описано множество, которое спецификатор изображает:

'abc' — любой из символов-литер 'a', 'b', 'c',

('abc') — любой терм, за исключением символов-литер 'a', 'b', 'c',

('a')L — любая буква, за исключением буквы 'a',

('a')L('0')N — любая буква, за исключением буквы 'a', или любая цифра, за исключением цифры '0'.

Имена спецификаторов описываются с помощью ключевого слова «S»: [ID] S [Sp], где [ID] — имя спецификатора без ограничителей «:»; [Sp] — спецификатор. Например, ADDOP S '+' — или MULTOP S '\*' означает, что имена ADDOP и MULTOP: можно употреблять в качестве спецификаций и элементов других спецификаторов. Например, значением переменных "S:ADDOP:x" и "S:(ADDOP:)Y" может быть только '+' или '-'.

Если спецификатор задан для S- или W-переменной, то это означает, что значение переменной должно принадлежать множеству, которое описывает спецификатор. Если спецификатор задан для VE-переменной, то это означает, что каждый терм значения переменной, стоящий на нулевом уровне скобочной структуры, должен удовлетворять спецификатору. Например,  $E(+ -)x$  — последовательность (может быть пустая) из литер '+' и '-';  $E(B)x$  — выражение вида  $(E_1)(E_2) \dots (E_N)$ , а  $S(L)XE(LD)Y$  — идентификатор;  $E((+ -))x$  — выражение, которое не содержит на нулевом уровне скобок ни одной литеры '+' или '-'. Например, в качестве значения для него подходит выражение вида  $(+)'(-)$ , но не  $+'(-)$ . Если у переменной несколько вхождений в ле-

вую часть, то у каждого вхождения может быть своя спецификация и значение каждого вхождения должно удовлетворять спецификации этого вхождения. Таким образом, получается, что множество допустимых значений переменной — это пересечение множества допустимых значений ее вхождений. Например,  $S(('a'))xS(('b'))x = Sx$  равносильно  $S(('ab'))xSx = Sx$ . Все спецификации, которые заданы для вхождений переменных в правую часть предложения, игнорируются.

На использование имен спецификаторов наложено следующее ограничение: если имя некоторого спецификатора используется при описании другого спецификатора, то оно должно быть описано раньше. Благодаря этому ограничению запрещаются циклические определения.

Употребление переменных в левых и правых частях предложений — мощное изобразительное средство. Теперь, чтобы решить, применимо ли предложение  $[L] = [R]$  к объектному выражению  $[ARG]$ , Рефал-машина должна определить, можно ли вместо переменных, входящих в  $[L]$ , подставить такие значения, чтобы получившееся объектное выражение совпало с  $[ARG]$ . При этом значения переменных должны быть допустимыми, т. е. соответствовать их типам и спецификациям. Кроме того, все вхождения одной и той же переменной должны заменяться на одно и то же значение. Описанное выше действие Рефал-машины по подбору значений переменных называется *синтаксическим отождествлением*. Если отождествление левой части  $[L]$  с  $[ARG]$  возможно, предложение является применимым, в противном случае — неприменимым.

При использовании переменных Рефал-машина на каждом шаге просматривает описание функции и находит самое первое применимое предложение, заменяет ведущий функциональный терм на правую часть этого предложения, предварительно подставив в нее вместо переменных те значения, которые получили эти переменные в результате синтаксического отождествления. Если окажется, что все предложения функции неприменимы, Рефал-машина сообщает, что «отождествление невозможно», и останавливается. Разумеется, в правой части предложения разрешается использовать только такие переменные, которые входят в левую часть. Кроме того, все вхождения одной и той же переменной в левую и правую части должны иметь одинаковый указатель типа переменной. Поскольку правые части предложений могут содержать функциональные скобки, можно описывать функции в терминах других функций или применять рекурсию.

Опишем, например, функцию SYMM, принимающую значение 'T', если аргумент — симметричное выражение, и значение 'F', если аргумент — несимметричное выражение:

$SYMM = 'T'$

$Sx = 'T'$

$Sx \text{ Ea } Sx = k/SYMM/Ea.$

$(Ea) = k/SYMM/Ea.$

$( )Ea( ) = k/SYMM/Ea.$

$(Wx \text{ E1 } Ea \text{ (E2 Wy)} = k/SYMM/Wx \text{ (E1) } Ea \text{ (E2) } Wy.$

$Ea = 'F'$

Будем говорить, что некоторая переменная — VE-переменная, если она является V-переменной или E-переменной. Если левая часть предложения содержит несколько VE-переменных, то может быть несколько вариантов приписывания переменным значений, приводящих к отождествлению левой части предложения с аргументом функции. Пусть, например, в поле памяти находится функция

$F \text{ E1 } ' ; ' \text{ E2} = k/G/E1. \text{ k/F/E2.}$

а в поле зрения — выражение

$k/F'al := a2; b1 := b2; c1 := c2'.$

Это выражение может быть отождествлено с левой частью так, что переменные "E1" и "E2" примут значения  $a1:=a2$  и  $b1:=b2$ ;  $c1:=c3$  соответственно. Возможен и другой вариант отождествления, при котором эти же переменные примут значения  $a1:=a2$ ;  $b1:=b2$  и  $c1:=c2$ . Следовательно, необходимо договориться, как поступает Рефал-машина при такой неоднозначности. Для устранения неоднозначности в Рефале могут использоваться два метода: отождествление слева направо и отождествление справа налево. При отождествлении слева направо Рефал-машина выбирает тот вариант, при котором первая слева VE-переменная принимает самое короткое значение. Если это не устраняет неоднозначности, то такой же отбор производится по второй слева VE-переменной, затем — по третьей слева и т. д. В нашем примере будет выбран первый из двух способов отождествления. При отождествлении справа налево Рефал-машина выбирает тот вариант отождествления, при котором первая справа VE-переменная принимает самое короткое значение. Если это не устраняет неоднозначности, то такой же отбор производится по второй справа VE-переменной, затем по третьей справа и т. д. В нашем примере будет выбран второй из двух способов отождествления.

Для отождествления справа налево необходимо поместить перед левой частью предложения ключевое слово "R", для отождествления слева направо — ключевое слово "L". Если направление отождествления не указано явно, Рефал-машина выполняет отождествление слева направо.

В простейшем случае Рефал-машина имеет поле памяти и поле зрения. В действительности у нее имеются и другие виды памяти. Один из них — «копилка», доступ к которой возможен с помощью специальных первичных функций, предназначенных для перемещения выражений из поля зрения в «копилку» и обратно.

Объектами обработки для программ, написанных на Рефале, являются выражения. Выражение представляет собой по существу способ представления древовидных структур в виде одномерных цепочек символов и скобок. При решении некоторых задач, однако, требуется обрабатывать более сложные структуры данных. В принципе, любые конструктивные объекты можно представить в виде деревьев, но это не всегда удобно, а иногда приводит к существенному замедлению работы программы. Средством Рефала, позволяющим обрабатывать произвольные графы, являются «ящики» — потенциально бесконечное множество запоминающих устройств. Каждый «ящик» содержит произвольное объектное выражение, которое может изменяться в процессе работы. Каждому «ящику» соответствует функция, с помощью которой можно получить доступ к его содержимому. Эти функции называются обменными. Таким образом, имеется взаимно однозначное соответствие между множеством обменных функций и множеством «ящиков».

Все «ящики» делятся на статические и динамические. Статические «ящики» существуют в течение всего времени выполнения программы и не могут ни порождаться, ни уничтожаться во время работы. Напротив, динамические «ящики» порождаются только во время работы и могут уничтожаться. Именно динамических «ящиков» являются символами-ссылками. Для работы с динамическими «ящиками» используются специальные первичные функции. При этом возможны ситуации, в которых имя «ящика» становится недоступным, так как ни в поле зрения, ни в «копилке», ни в других «ящиках» не сохранилось имен этих «ящиков». Поэтому дальнейшая работа программы не изменится, если эти «ящики» уничтожить. Ясно, что если обращаться к функциям работы с «ящиками» много раз, память Рефал-машины будет забиваться ненужными «ящиками». Конечно, для абстрактной Рефал-машины это не имеет никакого значения, ибо ее память потенциально бесконечна, но память реальной ЭВМ рано или поздно должна исчерпаться. Поэтому во всех реализациях Рефала, работающих с «ящиками», предусмотрен механизм сборки мусора. Сборка мусора автоматически запускается каждый раз, когда исчерпается свободная память. При этом обнаруживаются и удаляются все «ящики», к которым невозможно добраться прямо или косвенно из поля зрения, «копилки» или статических «ящиков».

## Модульность Рефал-программ

Часто бывает удобно разбить Рефал-программу на части, которые могут обрабатываться компилятором Рефала независимо друг от друга. Наименьшая часть Рефал-программы, которая может быть обработана компилятором независимо от других, называется *модулем*. Результат компиляции исходного модуля на Рефале представляет собой объектный модуль, который перед исполнением Рефал-программы должен быть объединен с другими модулями, полученными компиляцией с Рефала или других языков. Это объединение выполняется с помощью редакторов связей и загрузчиков. Детали зависят от используемой операционной системы.

Исходный Рефал-модуль должен начинаться с директивы START и кончаться директивой END:

```
[нмямод] START  
END
```

Имя модуля может быть опущено.

Функции, описанные в разных модулях, могут обращаться друг к другу. Если в некотором модуле используется функция, которая определена в другом модуле, эту функцию следует объявить внешней по отношению к данному модулю с помощью директивы EXTRN:

```
EXTRN [ф1], [ф2], ..., [фN]
```

Если в некотором модуле описана функция, к которой есть обращения из других модулей, эта функция должна быть объявлена входной точкой данного модуля с помощью директивы ENTRY:

```
ENTRY [ф1], [ф2], ..., [фN]
```

где [фi] — описание входной точки или внешней метки соответственно.

Рассмотрим одно из возможных решений задачи прокладки маршрутов между заданными городами, используя язык Рефал. Информацию о сети дорог будем представлять в поле зрения совокупностью «скобочных сумок» вида

(название-пункта-1 название-пункта-2 расстояние)

Предположим, что в нашем распоряжении имеются стандартные функции ввода (input) и вывода (output) информации. Тогда инициатор программы можно описать следующим Рефал-предложением:

```
way=k/route/k/net_of_roads/  
(k/input/"старт:" . " k/input/"финиш:"./0/)
```

В соответствии с алгоритмом функционирования Рефал-машины сначала будет активирована функция загрузки сети дорог (net\_of\_roads). В нашем случае загрузка описывается одним Рефал-предложением:

```
net_of_roads= ("Москва, Ленинград"/600/  
("Ленинград, Таллинн"/400/  
...  
("Москва, Одесса"/1400/)
```

Затем будет принята начальная и конечная точки маршрута и начнет работать функция поиска (route). Таким образом, к моменту начала работы функции поиска маршрута (route) в поле зрения имеется информация о сети дорог, загруженная в процессе работы функции net\_of\_roads, и, кроме того, аргументы функции route уже получили с помощью функции input определенное значение. Очевидно, что при поиске маршрута возможно три случая: между заданными пунктами имеется прямая дорога; дорога между заданными пунктами имеется, но проходит через промежуточные пункты; между заданными пунктами дороги нет. Каждой из этих альтернатив можно сопоставить отдельное Рефал-предло-

жение. Тогда описание функции route будет выглядеть следующим образом:

```
route ex(e1,"e2 sd)ey (e1","e2 s3)=k/output/  
    "расстояние:" k/add/(s3)sd..  
ex(e1","ea sd)ey (e1,e2 s3)=  
    k/route/ex ey(ea","e2 k/add/(s3)sd.).  
ex(e1","e2 s3)=k/output/"между городами" e1  
    "н" e2 "дороги нет".
```

Таким образом, вся Рефал-программа поиска маршрута состоит из трех основных функций (way, net\_of\_roads, route) и, естественно, отражает алгоритм поиска пути на графе, заданном матрицей смежности его вершин.

### Диалекты языка Рефал

Рассмотренная выше версия языка, если из нее исключить спецификаторы, представляет собой так называемый базисный Рефал. Работы по его теории и реализации на всех основных типах ЭВМ были завершены в середине 70-х годов [Климов и др., 1972; Климов и др., 1973; Турчин, 1974а; Базисный, 1977]. Новой идеей, породившей основной поток модификаций Рефала, стали спецификаторы. Будучи средством введения в язык концепции абстрактных типов данных, спецификаторы существенно повышают изобразительную мощь Рефала. Поэтому исследованию их было уделено самое серьезное внимание. В одном из ранних руководств по языку [Турчин, 1971] спецификаторы в полном объеме были введены в виде так называемых рекурсивных переменных и такой вариант Рефала получил название полного. Здесь спецификатор — обобщенное выражение, и действует он как вызов функции, в конечном счете вычисляющей значение рекурсивной переменной. В процессе такого вычисления аргумент функции, которая вычислялась до появления рекурсивной переменной, меняется на этапе синтаксического отождествления, что может привести к нежелательным побочным эффектам. Язык, в котором такие побочные эффекты были запрещены, получил название предикативного Рефала. Ни полный, ни предикативный Рефал реализованы не были.

Базисный Рефал [Базисный, 1977] включает ограниченный вариант спецификаторов, в рамках которого допускаются лишь явные перечисления элементов множества значений переменной символа. Однако в реализацию базисного Рефала эта важная в теоретическом отношении конструкция не вошла.

В 1976 г. появился Рефал-компилятор для ЭВМ БЭСМ-6 с входным языком, получившим название Рефал-2 [Климов и др., 1974]. Впоследствии он был реализован и на других типах ЭВМ [Введенков, 1975; Романенко, 1987а; Алешин и др., 1986]. Основным новшеством Рефал-2 было введение в язык и эффективная реализация ограниченных спецификаторов — конструкции, отвечающей нуждам практического программирования того времени (синтаксис и семантика ограниченных спецификаторов уже обсуждалась выше).

В связи с задачей выполнения эквивалентных преобразований на Рефале был предложен так называемый ограниченный Рефал [Турчин, 1972]. Здесь исключены переменные терма и, кроме того, открытые и повторные переменные типа выражения. Образец, удовлетворяющий таким ограничениям, называется L-выражением. Алгоритм отождествления для L-выражений существенно проще, чем для произвольных образцов, так как в нем нет перебора вариантов и для каждого фиксированного L-выражения образец изображается конечным деревом без циклов по объектному выражению. Благодаря этому ограниченный Рефал — более удобный объект для эквивалентных преобразований и в то же время трансляция из базисного Рефала в ограниченный трудностей не представляет. Каждому образцу соответствует множество объектных выражений. Поэтому можно говорить об образцах как о множествах, рассматривать над ними операции объединения и пересечения, ставить вопрос о выразимости результа-

тов этих операций в терминах образцов. При этом оказывается справедливой следующая теорема [Турчин, 1974б]. Пусть  $E$  — образец, а  $L$  —  $L$ -выражение. Тогда пересечение  $E$  с  $L$  можно представить в виде объединения  $E_1 \cup E_2 \cup \dots \cup E_n$ , где каждое  $E_i$  может быть получено из  $E$  некоторой подстановкой  $S_i$ , из  $L$  — некоторой другой подстановкой  $SL_i$ . При этом система подстановок  $\{S_i\}$  обладает тем свойством, что применение ее к произвольному  $L$ -выражению дает систему попарно не пересекающихся  $L$ -выражений. На этой теореме базируется мощное правило эквивалентных преобразований Рефал-программ, названное *прогонкой*. Суть его состоит в выполнении шага Рефал-машины над полем зрения, содержащим не какое-то определенное, а любое (из некоторого множества рабочих) выражение. При этом оказывается возможной глобальная оптимизация Рефал-программ за счет исключения «лишних» функций. Прогонка играет фундаментальную роль в проекте создания суперкомпилятора на основе языка Рефал [Турчин, 1986]. Проект практической системы для управляемых преобразований Рефал-программ на основе прогонки обсуждается в [Климов и др., 1987].

Для успешного осуществления прогонки базисный Рефал ограничивался, и это диктовалось необходимостью уметь выражать результаты прогонки в терминах исходного языка. При этом ограниченный Рефал был, по сути дела, собственной «неподвижной точкой». Но есть и другой путь достижения той же цели — расширение языка. Именно этот путь и привел в последнее время к созданию Рефал-4 [Романенко, 1987б], который является расширением Рефал-2 ровно до такой степени, чтобы получилась следующая «неподвижная точка», выдерживающая прогонку. Рефал-4 является собственной «неподвижной точкой» и при этом предоставляет более мощные изобразительные средства, чем полный Рефал [Турчин, 1971].

Одним из основных изобразительных средств, введенных в Рефал-4, являются так называемые *перестройки* — конструкции вида «источник»: «получатель», в которых источником служит выражение, а получателем — образец (в смысле Рефал-2). В момент исполнения перестройки некоторые из переменных, входящих в описание функции, уже получили значения, только они и могут входить в «источник». В «получатель» же могут входить любые переменные. Первый шаг в исполнении перестройки состоит в подстановке вместо переменных их значений, может быть, и вычисляемых. После окончания этого процесса «источник» превращается в выражение, не содержащее переменных, а «получатель» может содержать переменные, еще не получившие значений. Далее выполняется синтаксическое отождествление образца («получателя») с объектной цепочкой («источником»). Если существует несколько вариантов такого отождествления, все они запоминаются в стеке альтернатив Рефал-машины, переменные получают значения из первого варианта и выполнение программы продолжается. Однако если в дальнейшем какая-то часть программы потерпит неудачу, из стека альтернатив будет извлечена информация о других возможных значениях переменных и процесс повторится с точки «неуспеха».

Введение перестроек позволяет гибко управлять процессом отождествления, но предполагает, что все возможные варианты его будут отработаны. Вместе с тем уже в процессе выполнения одной части программы иногда становится ясно, что другие варианты к успеху не приведут. Тогда желательно удалить оставшиеся варианты из стека альтернатив и организовать дальнейшее выполнение программы без ненужных возвратов и удлинений открытых переменных. Для этого в Рефал-4 имеются специальные изобразительные средства разного уровня. Во-первых, здесь можно объединять общие части образцов в деревья с помощью конструкций ветвления. Во-вторых, введены специальные средства, специфицирующие возможность и/или необходимость возвратов, что позволяет управлять перебором вариантов отождествления.

Таким образом, Рефал выходит на новый виток в спираль своего развития. И можно предполагать, что ведущиеся здесь исследования приведут, с одной стороны, к созданию новых мощных версий этого языка, а с другой — дадут

дополнительный импульс в теории и практике эквивалентных преобразований и синтеза программ. Вместе с тем следует ожидать развития языка Рефал и по пути превращения его в практичный язык представления знаний.

## 1.6. Языки программирования интеллектуальных решателей

*В. Н. Лихолп, В. Н. Пильщиков*

Группа языков, которые можно назвать языками интеллектуальных решателей, в основном ориентирована на такую подобласть искусственного интеллекта, как решение проблем (problem solving), для которой характерны, с одной стороны, достаточно простые и хорошо формализуемые модели задач, а с другой — усложненные методы поиска их решения. Поэтому основное внимание в этих языках было уделено введению мощных структур управления, а не способам представления знаний. В данном параграфе будут рассмотрены языки PLANNER, CONNIVER, QA-4 и QLISP.

Основное внимание будет уделено языку PLANNER, для других языков рассмотрены лишь наиболее важные их отличия от языка PLANNER.

### Язык PLANNER

Этот язык можно считать родоначальником языков искусственного интеллекта, поскольку он дал толчок мощному языкотворчеству в этой области. Язык разработан в Массачусетском технологическом институте (MIT) в 1967—1971 гг. [Hewitt, 1973]. Вначале это была надстройка над Лиспом, в таком виде язык реализован на MacLisp под названием MICRO PLANNER [Sussman et al., 1971] и использован при создании известной системы SHRDLU [Winograd, 1972]. В дальнейшем PLANNER был существенно расширен и превращен в самостоятельный язык. В достаточно полном объеме он реализован в Шотландии под названием POPLER-1.5 [Davies et al., 1973], и в СССР под названиями Плэнер-БЭСМ [Пильщиков, 1982] и Плэнер-Эльбрус [Лихолп, 1985а], ввел в языки программирования много новых идей: автоматический поиск с возвратами (backtracking), поиск данных по образцу, вызов процедур по образцу, дедуктивный механизм и др. В дальнейшем эти идеи были использованы в других языках.

В рассмотрении языка PLANNER будем придерживаться версии Плэнер-БЭСМ, а в конце отметим особенности версии Плэнер-Эльбрус.

В качестве своего подмножества Плэнер содержит практически весь язык Лисп (с некоторыми модификациями) и во многом сохраняет его специфические особенности. Структура данных (выражений, атомов и списков), синтаксис программ и правила их вычисления в Плэнере аналогичны лисповским. Для обработки данных в Плэнере в основном используются те же средства, что и в Лиспе: рекурсивные и блочные (PROG и т. п.) функции. Практически все встроенные функции Лиспа, в том числе и функция EVAL, включены в Плэнер. Аналогично определяются новые функции. Как и в Лиспе, с атомами могут быть связаны списки свойств. Для иллюстрации лисповских черт Плэнера приведем рекурсивное определение функции MEMBER от двух аргументов E и L, которая проверяет, равен ли E хотя бы одному из элементов верхнего уровня списка L:

```
[DEFINE MEMBER (LAMBDA (E L)
  [COND ([EMPTY L] ())
        ([EQ [1 L] E] T)
        (T [MEMBER E [REST 1 L]])]])]
```

Выражения [EMPTY L], [1 L] и [REST 1 L] эквивалентны лисповским (NULL L), (CAR L) и (CDR L). Этот пример показывает не только близость Плэнера к Лиспу, но и отличия от него. Отметим некоторые из них. В Лиспе

при обращении к переменной указывается только ее имя, например  $X$ , сам же атом  $X$  как данное указывается так: (QUOTE  $X$ ) или  $"X"$ . В Плэнере используется обратная нотация: атомы обозначают самих себя, а при обращении к переменным перед их именем ставится префикс. Например,  $X$  — это сам атом  $X$ , а  $.X$ ,  $*X$  и  $!X$  — это обращения к переменной с именем  $X$ . При этом префикс указывает, как должна быть использована переменная. К примеру, префикс  $"."$  указывает, что надо взять значение переменной, а префикс  $"*"$  — присвоить ей новое значение. Такая нотация очень удобна для образцов, которые широко используются в Плэнере. Отличается от лисповского и синтаксис обращения к функциям, которое в Плэнере записывается в виде списка не с круглыми, а с квадратными скобками. Это связано с тем, что в Плэнере используется особое правило вычисления списков с круглыми скобками: значением такого списка является список из значений элементов исходного списка. Так, при значениях  $A$  и  $(B\ C)$  у переменных  $X$  и  $Y$  в результате вычисления  $(X\ (E\ Y)\ [!Y])$  будет получен список  $(A\ (E\ (B\ C)))\ B)$ .

В связи с этим правилом в Плэнер введены *сегментные выражения*, значениями которых являются последовательности данных (сегменты). При том же значении переменной  $Y$  сегментное обращение к ней  $!Y$  даст значение  $(B\ C)$ . Аналогично при сегментном обращении к функции, которое записывается в угловых скобках, у списка, являющегося значением функции, отбрасываются внешние скобки. Поэтому при том же значении переменной  $Y$  в результате вычисления  $(!Y\ Y < REST\ ! >)$  будет получен список  $(B\ C\ (B\ C)\ C)$ .

В сочетании с сегментными выражениями указанное выше правило вычисления списков с круглыми скобками становится простым и единообразным способом построения любых новых списков, позволяющим обойтись без лисповских функций LIST, CONS и APPEND, например лисповским выражениям  $(LIST\ X\ Y)$ ,  $(CONS\ X\ Y)$  и  $(APPEND\ X\ Y)$  соответствуют в Плэнере  $(.X\ .Y)$ ,  $(.X\ !.Y)$  и  $(!X\ !.Y)$ . В Плэнере легко реализуются и более сложные случаи построения списков, например эквивалентом лисповского  $(APPEND\ X\ (CONS\ (LIST\ Y\ Z)\ W))$  является  $(!X\ (.Y\ .Z)\ !.W)$ .

Обобщены в Плэнере и операции выделения и удаления элементов из списков. Например,  $[5\ .X]$  — это выделение пятого элемента от начала списка  $X$ , а  $[-5\ .X]$  — пятого от конца. При вычислении же  $[REST\ 3\ .X]$  будет получен список  $X$  без трех первых элементов, а при вычислении  $[REST\ -3\ .X]$  — без трех последних.

В Плэнере для обработки данных используются не только функции, но и *образцы* — конструкции, подобные тем, что используются в языке Рефал. Образцы в наглядной и лаконичной форме описывают правила анализа и декомпозиции данных, поэтому их применение облегчает написание программ и сокращает их тексты.

Синтаксически образец — это выражение, а содержательно он рассматривается как шаблон, который накладывается на другое анализируемое выражение, чтобы определить, имеет ли оно нужную структуру. Процесс такого наложения называется сопоставлением (pattern-matching). В Плэнере его осуществляет функция IS, фиксирующая своим значением истина-ложь исход сопоставления. Обычно образец задается в виде списка, элементы которого указывают, какие должны находиться в соответствующих позициях анализируемого выражения (оно также должно быть списком). Если элемент образца — атом, обращение к переменной с префиксом  $"."$  или простое обращение к функции, то соответствующий элемент выражения должен равняться этому атому, значению этой переменной или функции. Если элемент образца — подсписок, то противостоящий ему элемент выражения также должен быть подсписком со структурой, соответствующей этому подобразцу. Например, образец  $(ЦЕНА\ .X\ ([+ .K\ !]\ РУБЛЕЙ))$  требует, чтобы сопоставляемое с ним выражение было списком из трех элементов: первый — атом ЦЕНА, второй совпадает со значением переменной  $X$ , а третий является списком из двух элементов, первый из которых равен увеличенному на 1 значению переменной  $K$ , а второй — атом РУБЛЕЙ.



Элементами образца могут быть также обращения к переменным с префиксом «\*». Они сопоставимы с чем угодно, причем как побочный эффект сопоставления этим переменным присваиваются новые значения — сопоставляемые с ними элементы выражения. Например, при сопоставлении

[IS (ЦЕНА \*ПРЕДМЕТ \*ЦЕНА)  
(ЦЕНА КНИГА (5 РУБЛЕЙ))]

переменная ПРЕДМЕТ получит значение КНИГА, а переменная ЦЕНА — значение (5 РУБЛЕЙ). Таким образом, с помощью образцов можно не только проверять структуру выражений, но и выявлять заранее неизвестные части выражений, присваивая их переменным. Однако такие присваивания выполняются лишь тогда, когда сопоставление удачно в целом.

В рассмотренных примерах каждый элемент образца сопоставлялся только с одним элементом анализируемого выражения. Однако в образцах-списках допускается использование и сегментных подобразцов, которые сопоставляются сразу с несколькими (соседними) элементами выражения. К ним, в частности, относятся обращения к переменным с префиксами "1." и "1\*". Например, подобразцу 1.X соответствует сегмент, совпадающий с последовательностью элементов списка, который является значением переменной X, а подобразцу 1\*X соответствует любой сегмент, причем в случае удачного исхода сопоставления в целом переменная X получит новое значение, которым становится список из элементов этого сегмента. Так, при значении (БЕЛЫЙ ШАР) у переменной X сопоставление

[IS (1.X 1\*Y НАХОДЯТСЯ НА 1\*Z)  
(БЕЛЫЙ ШАР И СИНИЙ КУБ  
НАХОДЯТСЯ НА СТОЛЕ)]

удачно, причем переменная Y получит значение (И СИНИЙ КУБ), а переменная Z — значение (СТОЛЕ).

С помощью рассмотренных образцов можно осуществлять довольно сложный анализ выражений, однако в общем случае таких образцов недостаточно. Например, с их помощью (не привлекая функции) нельзя проверить, являются ли анализируемое выражение списком из чисел. Чтобы в виде образцов можно было описать любые правила анализа, в Плэнер введены особые процедуры, названные *сопоставителями* (ACTORS). Обращение к ним записывается так же, как и обращение к функциям, однако к сопоставителю можно обратиться только в образцах и его выполнение заключается не в вычислении какого-либо значения, а в проверке, обладает ли сопоставляемое с ним выражение определенным свойством.

В Плэнере имеются встроенные сопоставители, которые проверяют простейшие свойства и задают базовые способы объединения образцов. Например, сопоставитель NUM (без аргументов) проверяет свойство «быть числом», поэтому образцу [NUM], являющемуся обращением к этому сопоставителю, соответствуют только числа. Следовательно, сопоставление [IS ([NUM]+[NUM]).X] удачно, если переменная X имеет, скажем, значение (5+6), и неудачно при значении (5+A). Сопоставитель LIST проверяет свойство «быть списком», причем его аргумент накладывает ограничение на длину анализируемого списка. Например, образцу [LIST 2] соответствует любой список из двух элементов, а сегментному варианту этого образца <LIST 2> соответствует сегмент из двух любых элементов, поэтому сопоставление [IS ([LIST 2] <LIST 2>)] ((A B) C D)] удачно.

Примерами встроенных сопоставителей другой группы могут служить сопоставители AUT и NON, определяющие соответственно дизъюнкцию и отрицание образцов. Сопоставителю AUT, аргументами которого являются образцы, соответствует любое выражение, если оно соответствует хотя бы одному из этих образцов. Например, образцу [AUT [NUM] [LIST 2]] соответствует либо число, либо список из двух элементов. Сопоставитель NON, имеющий один аргумент-образец, соответствует любому выражению, если оно не соответствует

данному образцу. Например, сопоставление [IS ([NON A] B) X] удачно, скажем, при значении (B B) у переменной X и неудачно при значении (A B). На основе встроенных сопоставителей можно определять новые сопоставители для проверки любых желаемых свойств. Например, для проверки свойства «быть списком из чисел» можно определить следующий сопоставитель:

```
[DEFINE NUMS-LIST (KAPPA ( )
  [AUT ( ( ) ([NUM] <NUMS-LIST>))])]
```

Сопоставители определяются так же, как функции, только их определяющее выражение начинается с ключевого слова KAPPA, а в качестве тела указывается образец. В данном случае это обращение к сопоставителю AUT, который требует, чтобы анализируемое выражение было либо пустым списком, либо списком, первый элемент которого — число, а оставшиеся элементы образуют сегмент из чисел.

Рассмотренное подмножество Плэнера можно использовать независимо от других его частей: оно представляет собой мощный язык программирования, удобный для реализации различных систем символьной обработки. Остальные части Плэнера ориентируют его на область искусственного интеллекта, предоставляя средства для описания задач (исходных ситуаций, допустимых операций, целей), решения которых должна искать система искусственного интеллекта, реализуемая на Плэнере, и средства, упрощающие реализацию процедур поиска решения этих задач.

*Базой данных* в Плэнере называется совокупность утверждений, описывающих обстановку, в которой система искусственного интеллекта решает свою задачу. Синтаксически утверждение — это список, а содержательно — это описание какого-то одного истинного факта обстановки. Все вместе они описывают обстановку в целом.

Для работы с базой данных используются следующие основные операции: поиск по образцу и запись, вычеркивание утверждений. Поиск по образцу (pattern-directed retrieval) применяется для анализа содержимого базы данных: задается некоторый образец и в базе данных отыскивается утверждение, соответствующее этому образцу. Такой поиск осуществляет функция SEARCH. Например, [SEARCH (ON A B)] — это поиск утверждения, равного списку (ON A B), что содержательно можно интерпретировать как проверку, находится ли кубик A на кубике B, поиск [SEARCH (ON \*X C)] можно интерпретировать как определение, находится ли что-нибудь на кубике C, и, если да, то что именно, как побочный эффект переменной X присвоится название кубика, находящегося на C.

Запись новых утверждений в базу данных осуществляет функция ASSERT, а вычеркивание утверждений из базы данных — функция ERASE. Например, [ASSERT (ON A C)] запишет в базу данных утверждение (ON A C), а [ERASE (ON A B)] удалит утверждение (ON A B). С помощью этих операций моделируются изменения в обстановке, которую описывает база данных: например, указанными действиями имитируется перенос кубика A с кубика B на кубик C.

При изменении базы данных надо следить за тем, чтобы сохранялось непротиворечивое описание обстановки. Так, если в базу данных записывается утверждение (ЖЕНАТ ПЕТР), то одновременно из базы данных должно быть удалено утверждение (ХОЛОСТ ПЕТР). Чтобы упростить контроль за базой данных, в Плэнер введены записывающие (антеседельные) и вычеркивающие теоремы. *Теорема* — это процедура, содержащая помимо тела еще и образец, на основе которого она вызывается. Записывающие теоремы автоматически вызываются тогда, когда в базу данных записываются утверждения, соответствующие их образцам, а вычеркивающие теоремы — при вычеркивании утверждений из базы данных. Встроенным теорем в языке нет, поэтому все теоремы надо определять. Пример записывающей теоремы:

```
[DEFINE БРАК (ANTEC (X) (ЖЕНАТ *X)
  [ERASE (ХОЛОСТ X)])]
```

Как и другие процедуры языка (функции, сопоставители), теоремы имеют имена, но эти имена обычно не используются. Определяющее выражение записывающей теоремы начинается со слова ANTES (вычеркивающей — с ERASING), за которым следует список ее локальных переменных. Далее указывается образец теоремы, в нашем случае — это (ЖЕНАТ \*X), и тело теоремы, в общем случае совпадающее с телом функций PROG, состоит лишь из обращения к функции ERASE. Записывающая теорема вызывается всякий раз, когда в базу данных записывается утверждение о том, что кто-то женат. Например, при записи утверждения (ЖЕНАТ ПЕТР) вызов теоремы начинается с сопоставления ее образца с этим утверждением, в результате переменная X получает значение ПЕТР. Затем вычисляется тело теоремы, что в нашем случае приводит к вычеркиванию из базы данных утверждения (ХОЛОСТ ПЕТР). На этом вычисление теоремы завершается. Если имеются и другие теоремы, образцы которых соответствуют записанному утверждению, то все они вызываются по очереди. Каждая из них осуществляет свой контроль за изменением базы данных.

Отметим, что теоремы Плэнера — это пример процедурного представления знаний, т. е. представления их не в декларативной форме (таковой, например, является база данных), а в виде процедур. Наиболее наглядно это проявляется в основном типе теорем Плэнера — в целевых (консеквентных) теоремах, которые, как и база данных, используются для описания условий задачи, решаемой интеллектуальной системой. Дело в том, что одной базы данных недостаточно для полного описания этих условий, поскольку в виде утверждений можно описать только конкретные факты о конкретных объектах, а надо еще описывать отношения между используемыми понятиями (скажем, «между», «слева» и «справа») и те операции, которые разрешено применять при решении задачи. Для этого и используются целевые теоремы.

Структура и правила вычисления целевых теорем такие же, как у других теорем (только их определяющее выражение начинается со слова CONSEQ), но вот условия, при которых они вызываются, иные. В программе задается некоторый образец и вызывается любая целевая теорема, образец которой соответствует этому вызывающему образцу. Такой вызов по образцу (pattern-directed invocation) используется в Плэнере следующим образом. Каждая целевая теорема (далее просто «теорема») описывает некоторое действие, причем результат этого действия, т. е. та цель, которой можно достичь с помощью теоремы, описывается в виде ее образца, а тело теоремы описывает собственное действие, т. е. то, что надо сделать для достижения данной цели. В виде же вызывающего образца описывается цель, достижение которой желательно на очередном шаге программы. В этих условиях вызов теорем по образцу — это автоматический поиск средств для достижения поставленной цели.

Пронлюстрируем сказанное на примерах. Рассмотрим снова пример с кубиками. Понятие «X расположен выше Z» определяется так:

$$\begin{aligned} \text{ON}(X, Z) &\rightarrow \text{ABOVE}(X, Z) \\ \text{ON}(X, Y) \text{ U } \text{ABOVE}(Y, Z) &\rightarrow \text{ABOVE}(X, Z) \end{aligned}$$

Такие предложения обычно интерпретируются слева направо: если X находится на Z, тогда X выше Z; если X находится на Y и при этом Y выше Z, то X также выше Z. Однако возможна иная, обратная, интерпретация: если надо доказать, что X выше Z, то для этого надо в первом случае проверить, находится ли X на Z, а во втором — сначала найти Y, на котором находится X, а затем показать, что Y расположен выше Z. Именно эта императивная интерпретация положена в основу описания логических предложений в виде теорем: то, что надо доказать (проверить, найти), выносится в образец теоремы, а те действия, которые должны быть выполнены для этого, оформляются как тело теоремы. С учетом этого получаем следующие теоремы:

$$\begin{aligned} \text{[DEFINE TH1 (CONSEQ (X Z)} \\ \quad \quad \quad \text{(ABOVE *X *Z)} \\ \text{[SEARCH (ON .X .Z)]})]} \end{aligned}$$

```
[DEFINE TH2 (CONSEQ (X Y Z)
                    (ABOVE *X *Z)
                    [SEARCH (ON .X *Y)]
                    [GOAL (ABOVE .Y .Z)])]
```

Использованная здесь функция GOAL осуществляет вызов теорем по образцу, являющемуся ее аргументом. Предположим, что база данных содержит утверждения (ON A B), (ON B C) и (ON C TABLE), т. е. описывает башню из трех кубиков, и что мы хотим найти нечто W, выше которого расположен кубик A. Такая цель задается в виде образца (ABOVE A \*W), поэтому нам надо вычислить [GOAL (ABOVE A \*W)].

Имеются две теоремы, образцы которых соответствуют этому вызываемому образцу. Функция GOAL по своему усмотрению вызывает любую из них. Пусть вызвана теорема TH1. Тогда при сопоставлении ее образца с вызывающим образцом переменная X получит значение A, а переменная Z останется без значения, но будет отождествлена с переменной W. Далее вычисляется тело теоремы, т. е. функция SEARCH, которая найдет в базе данных утверждение (ON A B), соответствующее ее образцу (ON A .Z), и присвоит переменной Z значение B (если Z не имеет значения, то образец .Z ведет себя как образец \*Z), которое автоматически будет присвоено переменной W. На этом вычисление теоремы и функции GOAL завершается. Значение B у переменной W и будет ответом на поставленный вопрос. Если же функция GOAL вызовет теорему TH2, тогда функция SEARCH из этой теоремы также найдет утверждение (ON A B) и присвоит переменной Y значение B, а затем функция GOAL поставит новую цель (ABOVE B .Z), аналогичную исходной. Если для достижения этой цели будет вызвана теорема TH1, то Z получит значение C, которое станет и значением переменной W, что и будет другим ответом на исходный вопрос.

Рассмотренные теоремы, описывающие отношения между разными понятиями, не меняют базу данных. Но в Плэнере допускаются и теоремы, которые ради достижения поставленной цели меняют базу данных. Если в текущей базе данных цель не является истинной, тогда они меняют базу данных так, чтобы в новой базе данных цель стала истинной. При этом, конечно, нельзя менять базу данных произвольно, а только «законными» способами, согласно «правилам игры». Именно эти «законные» операции и описываются данными теоремами. Например, операцию переноса кубика X из башни на стол можно описать следующей теоремой:

```
[DEFINE TRANSFER (CONSEQ (X Y Z) (ON X TABLE)
                        [IF ([SEARCH (ON *Y .X)] [GOAL (ON .Y TABLE)])]
                        [SEARCH (ON .X *Z)]
                        [ERASE (ON .X .Z)]
                        [ASSERT (ON .X TABLE)])]
```

Результат этой операции (то, что X окажется на столе) вынесен в образец теоремы. Тело ее описывает следующие действия. Если на X находится некий кубик Y, то ставится цель — сделать так, чтобы Y оказался на столе, т. е. чтобы на X ничего не было, так как только в этом случае можно на законном основании переносить X. Если данная цель достигнута, а для этого может быть вызвана любая подходящая теорема, в том числе и наша, или если на X с самого начала ничего не было, тогда определяется Z, на котором находится X, и затем «выполняется» перенос: из базы данных удаляется утверждение о том, что X находится на Z, и записывается утверждение о том, что X теперь на столе.

Если снова рассмотреть башню из кубиков A, B и C (A сверху) и поставить целью сделать так, чтобы B оказался на столе, т. е. вычислить [GOAL (ON B TABLE)], то будет вызвана теорема TRANSFER, которая сначала при рекурсивном обращении к себе перенесет на стол верхний кубик A, а затем и кубик B. В конце концов в базе данных окажутся утверждения

(ON A TABLE), (ON B TABLE) и (ON C TABLE), а значит, исходная цель достигнута.

Из приведенных примеров видно, что на Пленере можно программировать, описывая то, что имеется и что надо получить, без явного указания, как это делать (впоследствии этот стиль программирования лег в основу языков спецификаций — Пролога и др.). Ответственность же за поиск решения описываемой задачи берет на себя встроенный в язык дедуктивный механизм (механизм автоматического достижения целей), в основе которого лежит вызов теорем по образцу. Однако только вызова теорем по образцу недостаточно для такого механизма. Для достижения поставленной цели может подойти несколько теорем, которые предлагают разные способы достижения этой цели, например теоремы TH1 и TH2. Следовательно, надо уметь выбирать из нескольких альтернатив одну. Кроме того, если выбранная альтернатива оказалась unsuccessful (например, теорема не смогла достичь цели), тогда надо суметь вернуть программу назад и изменить ранее сделанный выбор. Таким образом, нужен механизм перебора, и такой механизм — режим возвратов (backtracking) — введен в язык.

Суть работы программы в режиме возвратов в следующем. В программе допускаются «развилки» — процедуры, которые имеют несколько вариантов своей работы. Такова, например, функция [SEARCH (ON \*X \*Y)], образуемой которой может соответствовать несколько утверждений базы данных, или функция [GOAL (ABOVE A C)], которая может вызвать теорему как TH1, так и TH2. Не зная заранее, какая из альтернатив приведет к успеху, программа выбирает одну из них, а другие пока запоминает. Сделав выбор, программа продолжает свои вычисления — исследует выбранный вариант. В некоторый момент может быть установлено, что программа зашла в тупик и не может продолжить дальше вычисления. Например, при вызове теоремы TH1 по образцу (ABOVE A C) функция SEARCH из ее тела не сможет найти в базе данных утверждения (ON A C). В этом случае вырабатывается специальный сигнал неуспеха, по которому программа автоматически возвращается назад к последней развилке, чтобы здесь исправить ранее сделанный выбор. Причем при возврате уничтожаются все следы работы программы на неуспешной ветви вычислений: восстанавливаются прежние значения переменных, прежнее состояние базы данных и т. д. Тем самым программа возвращается в такое состояние, как будто она и не делала неуспешного выбора.

Вернувшись к развилке, программа выбирает новую альтернативу и с этого места возобновляет свои вычисления — исследует новый вариант. Если и он оказался неуспешным, снова происходит возврат к развилке, где выбирается следующая альтернатива, и т. д. В конце концов либо будет найден успешный вариант, либо будут исчерпаны все альтернативы развилки. Во втором случае происходит возврат к предыдущей развилке, если она есть, чтобы теперь уже здесь сделать новый выбор.

Выполнение программы в режиме возвратов удобно для ее автора тем, что язык берет на себя ответственность за запоминание развилок и оставшихся в них альтернатив, за осуществление возвратов к ним и восстановление прежнего состояния программы — все это делается автоматически. Однако такой автоматизм не всегда выгоден, так как в общем случае он ведет к «слепому» перебору. И может оказаться так, что при вызове теорем наиболее подходящая из них будет вызвана последней, хотя автор программы заранее знает о ее достоинствах. Учитывая это, Пленер предоставляет средства управления режимом возвратов. Например, разрешено давать рекомендации функции GOAL, какие теоремы и в каком порядке она должна вызывать. Так, в обращении

[GOAL (ЛЮБИТ \*X МАША) (USE TH5 [NON TH3])]

рекомендация (она начинается с USE) требует, чтобы из всех теорем с подходящими образцами в первую очередь была вызвана теорема TH5, а если она окажется неуспешной, тогда любые другие подходящие теоремы, но только не теорема TH3. Такие рекомендации, а также иные средства управления режи-

мом возврата (уничтожение развилок, возврат по неучасу не к последней, а сразу к одной из предыдущих развилок и т. п.) позволяют автору программы взять под свой контроль работу дедуктивного механизма.

Описанная выше версия языка Плэнер реализована в 1976 г. в МГУ на ЭВМ БЭСМ-6. Система Плэнер-БЭСМ [Пильщиков, 1982] интерпретирующего типа. На языке Плэнер-БЭСМ реализованы известный лингвистический процессор TULIPS-2 [Мальковский, 1985], система ЛУЧ, обучающая языку Лисп [Большакова и др., 1986], и ряд других систем. С появлением MBK Эльбрус была предпринята попытка реализовать этот язык на данной машине. Оказалось, что многие возможности языка очень хорошо «накладываются» на аппаратные механизмы MBK, однако режим возвратов эффективно реализовать не удастся. Поэтому было решено отказаться от этого режима, а взамен ввести параллельные процессы. Полученная версия языка была названа Плэнер-Эльбрус.

Запуск параллельных процессов в этой версии осуществляет функция [PAR K E1 ... EN], где E1 — выражения, которые должны быть вычислены параллельно; аргумент K — это степень ветвления, который задает максимальное число активных параллельных процессов, Допустимое в данном месте программы. Если уже имеется K активных процессов, тогда функция PAR прекращает запуск новых процессов до тех пор, пока не будет завершен хотя бы один из активных. Например, при вычислении [PAR 2 [F1] [F2] [F3]] будет сначала инициализировано параллельное вычисление функции F с аргументами 1 и 2, а вычисление [F3] начнется только по завершении одного из этих двух процессов. Изменяя степень ветвления, можно организовать различные стратегии перебора: при степени 1 происходит «перебор вглубь», что аналогично режиму возвратов, при больших степенях — «перебор вширь», а при промежуточных значениях реализуются смешанные стратегии.

Функция PAR используется тогда, когда все указанные процессы должны быть безусловно инициализированы. Но это не всегда удобно, поэтому в язык введена функция [PARLOOP K X L P E], которая из множества потенциальных процессов запускает только некоторые. Для каждого элемента списка L выполняются следующие действия: этот элемент присваивается переменной K, затем вычисляется предикат P и, если он истинен, запускается процесс вычисления выражения E при данном значении X (аргумент K задает степень ветвления). Например, напечатать все решения известной задачи о 8 ферзях можно с помощью выражения [PACCTABЬ 1 ( )], где

```
[DEFINE PACCTABЬ (LAMBDA (HL)
  [COND ([EQ Н 9] [PRINT L])
    (Т [PARLOOP Н V (1 2 3 4 5 6 7 8)
      [НЕБЬЕТ V L]
      [PACCTABЬ [+ Н1] (L L V)]]))] ]
```

Здесь Н — номер горизонтали, на которой должен быть размещен очередной ферзь; L — список номеров вертикалей, в которые поставлены ферзи на предыдущих горизонталях. Если Н=9, то тем самым получена одна из расстановок, которая и печатается, иначе из 8 вертикалей горизонтали Н выбираются только те, которые не бьются предыдущими ферзями, и для каждой из них запускается процесс, пытающийся расставить ферзей на следующих горизонталях. В данном примере степень ветвления увеличивается с возрастанием номера горизонтали: чем ближе к концу, тем больше вариантов разрешается рассматривать одновременно.

Функции PAR и PARLOOP оканчивают свою работу либо после завершения всех инициализированных в них процессов, либо в случае выхода по глобальной метке одного из процессов. При таком выходе принудительно завершаются и все остальные процессы. Это полезно, когда какой-то процесс нашел решение и дальнейшая работа остальных уже не нужна.

В языке есть средства для синхронизации параллельных процессов. Например, функция [REGION I E] производит «захват» объекта I, выбранного для

синхронизации, на время вычисления выражения E; если этот объект уже был захвачен, то функция ожидает его освобождения. Например, для корректного увеличения на 1 значения переменной X, общей для нескольких процессов, надо выполнить [REGION X [SET X [+ .X 1]]]. Функция [WAIT S T] ожидает возникновения сигнала с именем S и завершает свою работу либо после его появления, либо по истечении времени T. Сигнал же можно возбудить функцией SIGNAL.

Поскольку в языке Плэнер-Эльбрус нет режима возвратов, то в нем нет и тесно связанных с этим режимом теорем; база данных сохранена, но изменены средства поиска утверждений: функция SEARCH ищет в базе данных только одно утверждение, а для просмотра всех утверждений, соответствующих образцу, используется функция LOOPDB, которая является аналогом генераторов языка CONNIVER.

Как показал опыт использования системы Плэнер-БЭСМ, при создании систем управления с элементами искусственного интеллекта возникает потребность в подключении модулей, написанных на других языках программирования. В системе Плэнер-Эльбрус [Лихолип, 1985б] такая возможность реализована. Кроме того, в этой системе помимо интерпретатора имеется и компилятор.

### Язык CONNIVER

Язык CONNIVER разработан в 1972 г. [McDermott, 1972], реализован как надстройка над языком MacLisp и использован при создании системы HACKER [Sussman, 1973]. Авторы языка CONNIVER выступили с критикой некоторых идей языка Плэнер. В основном она относилась к автоматическому режиму возвратов, который в общем случае ведет к неэффективным и неконтролируемым программам, особенно если они состоят из неквалифицированными пользователями, а также предлагает лишь одну из дисциплин управления, тогда как в интеллектуальных системах используются и другие. Авторы CONNIVER отказались от автоматического режима возвратов, считая, что встраивать в язык какие-то фиксированные дисциплины управления (кроме простейших — циклов, рекурсии) не следует и что автор программ должен сам организовывать нужные ему дисциплины управления, а для этого язык должен открыть пользователю свою структуру управления и предоставить средства работы с ней. Эта концепция была реализована в CONNIVER следующим образом.

При вызове процедуры в памяти отводится место, где хранится информация, необходимая для ее работы. Здесь, в частности, располагаются локальные переменные процедуры, указатели доступа (ссылка на процедуру, переменные которой доступны из данной) и возврата (ссылка на процедуру, которой надо вернуть управление). Обычно эта информация скрыта от пользователя, а в языке CONNIVER такой участок памяти (фрейм) открыт: пользователь может просматривать и менять содержимое фрейма. В языке *фреймы* представляют специальный тип данных, доступ к которым осуществляется по указателям. Для работы с фреймами используются следующие функции:

(FRAME) — выдает указатель на текущий фрейм;  
(ACCESS FR) — выдает указатель доступа из фрейма FR;  
(CONTROL FR) — аналогично для указателя возврата;  
(SETACCESS FR FR1) — делает указателем доступа в фрейме FR ссылку на фрейм FR1;  
(SETCONTROL FR FR1) — аналогично для указателя возврата.

С помощью этих функций пользователь получает доступ к любому существующему фрейму и может «подвешивать» любой фрейм к любому другому как по доступу, так и по возврату (рис. 1.7). В связи с этим легко реализуется, например, такая особенность языка Лисп, как FUNARG, когда внешние переменные функции берутся не из окружения вызова (что обычно для Лиспа), а из окружения, где функция была определена (что характерно для языков типа Алгол-60).

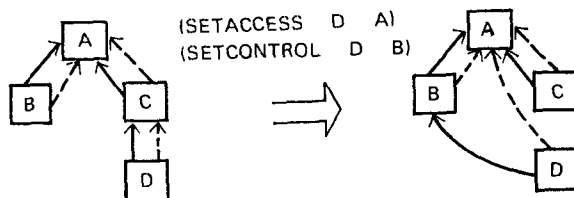


Рис. 1.7. Изменение связей между фреймами по доступу и управлению языка CONNIVER

Кроме того, в языке CONNIVER имеется еще один тип данных — теги. *Тег* — это адрес некоторой точки программы, который можно запомнить, а затем передать по нему управление из любого другого места программы (фрейм, на который есть хотя бы одна ссылка, всегда сохраняется в памяти). Для работы с тегами используются следующие функции:

(TAG *МЕТКА*) — выдать тег точки программы, что помечена указанной меткой;

(GO *ТЕГ*) — переход по тегу.

На основе указанных средств можно реализовать различные режимы работы программы. Например, режим сопрограмм, когда процедуры постоянно «перепрыгивают» друг другу управление, реализуется по следующей схеме:

```
(CDEFUN F ( )
  (CSETQ LF (TAG "F1"))
  (G)
:F1 . . . . .
  (CSETQ LF (TAG "F2"))
  (GO LF)
:F2 . . . . .)

(CDEFUN G ( )
  . . . . .
  (CSETQ LG (TAG "G1"))
  (GO LF)
:G1 . . . . .)
```

Здесь каждая из процедур, прежде чем передать управление другой, сообщает через некоторую переменную тег своей точки, с которой она должна будет возобновить свою работу. Примерно так же реализуется режим возвратов. Вводится специальный список-стек, куда каждая процедура-развилка, сделав выбор первой альтернативы, заносит тег той своей точки, где она выбирает следующую альтернативу, после чего процедура заканчивает свою работу. При появлении неуспеха из этого стека выбирается последний тег, и по нему делается переход. Аналогично реализуется и отмена побочных эффектов при возврате по неуспеху: процедура, производящая побочный эффект (например, меняющая состояние базы данных), должна иметь часть, отменяющую этот эффект. Когда процедура в первый раз заканчивает свою работу, то она заносит в специальный список тег этой «отменяющей» части. При неуспехе надо выбрать этот тег и передать по нему управление, в результате и произойдет отмена побочного эффекта.

В языке CONNIVER можно реализовать и другие режимы программы, однако, как показала практика, такая реализация — трудная и кропотливая работа, требующая высокой квалификации. В этом недостаток языка. Хотя пользователь и получает гибкие средства управления, одновременно на него ложится большая работа. Язык CONNIVER хорош не для реализации сложных систем; а как база, на основе которой квалифицированные программисты готовят нужные механизмы управления для других пользователей.

Учитывая сложность реализации дисциплин управления, авторы языка были вынуждены включить в него ряд фиксированных механизмов управления — генераторы и методы. *Генератор* — это аналог процедур-развилки языка Плэ-



иер. Но если в языке Плэнер разрыв между выбором альтернативы в развилке и ее анализом, а в случае необходимости выработкой неуспеха может быть сколь угодно велик, то в языке CONNIVER этот разрыв сведен к минимуму: обращаться к генераторам можно только в специальной функции, которая тут же проводит анализ альтернатив. Этим CONNIVER избавляется от негативных последствий глобальных возвратов по неуспеху, когда приходится отменять предыдущую работу чуть ли не всей программы. Что же касается методов, то это аналог теорем языка Плэнер, причем и для них приняты меры для локализации возврата по неуспеху.

В языке CONNIVER есть и база данных, но в отличие от Плэнера, она может существовать в нескольких экземплярах. При использовании базы данных как модели той обстановки, в которой система искусственного интеллекта решает задачи, нередко возникает потребность в нескольких таких моделях. Например, если система составляет планы действия робота на случай солнечной и дождливой погоды, то желательно иметь две модели, в одной из которых записано утверждение (СОЛНЕЧНО), а в другой — (ДОЖДЛИВО), и планировать в каждой из них независимо друг от друга. Учитывая это, в язык CONNIVER введены контексты базы данных. Программа начинает работу с одним контекстом, а затем может ввести новые. Появление нового контекста означает, что состояние текущего (родительского) контекста «замораживается» и теперь все изменения и поиск в базе данных будут осуществляться только в новом (дочернем) контексте (его начальное состояние совпадает с родительским). Закончив работу с «дочерним» контекстом, программа уничтожает его, и теперь текущим снова становится родительский контекст. На основе одного контекста можно определить несколько независимых дочерних контекстов, так что в общем случае существует целое дерево контекстов.

### Языки QA-4 и QLISP

Широко известна система QA-3 [Green, 1969], в которой для решения задач впервые использован метод резолюций. Язык QA-4, созданный в Стэнфордском исследовательском институте (SRI) в 1969—1971 гг. [Derksen, 1973], представляет собой воплощение идей этой системы в виде языка программирования. Язык QA-4 вобрал в себя практически все идеи языка PLANNER и развил их дальше. С некоторыми модификациями QA-4 был реализован на Interlisp и получил название QLISP [Sacerdoti, 1976]. На этом языке были написаны такие известные системы, как NOAH [Sacerdoti, 1975] и DEDALUS [Mapna, 1975].

Языки QA-4 и QLISP во многом повторяют языки PLANNER и CONNIVER, но в них есть ряд интересных особенностей. В языке QA-4 помимо списков, в которых учитывается как порядок расположения, так и повторяемость элементов, используются также *бэги* (bag) — неупорядоченные наборы, в которых учитываются дубли, и *множества* (set) — неупорядоченные наборы без копий. Наличие таких типов данных полезно, например, в следующей ситуации. Если указать, что совокупность аргументов некоторой функции является бэгом, то транслятор уже сам будет учитывать коммутативность этой функции. Характерной особенностью QA-4 является то, что разные, но равные выражения (списки и т. п.) имеют единое внутреннее представление. Это повышает эффективность выполнения многих операций над данными (например, поиск в базе данных) и позволяет связывать списки свойств не только с атомами, как в Лиспе, но и с любыми другими выражениями.

Далее, если в языке PLANNER функции, которые вызываются по имени, отличаются от теорем, вызываемых по образцу, то в языке QA-4 эти виды процедур объединены. При описании функции указывается не список формальных параметров, а образец (DEF F (LAMBDA образец тело)). Вызывать функции можно по имени (F аргумент), и тогда аргумент будет сопоставлен с образцом, а также по образцу (GOAL образец), и тогда подыскивается функция с подходящим образцом. Чтобы сократить поиск подходящей функции, в языке QA-4 функции можно разбить на группы и при вызове по образцу указывать ту груп-

пу, среди функций которой и будет вестись поиск. В QA-4 также используется режим возвратов, но его можно «включать» и «выключать». Например, при вызове по образцу может подойти несколько функций. Если в GOAL указан только образец, то будет вызвана только одна подходящая функция и при этом никакой развилки здесь не будет установлено, а вот при вызове (GOAL образец BACKTRACK) в этом месте программы будет установлена развилка, поэтому в случае неуспеха GOAL будет искать другую функцию.

В QA-4 развита идея записывающих и вычеркивающих теорем языка PLANNER в виде так называемых демонов. Демон — это процедура, которая автоматически вызывается при наступлении определенного события в программе, например при изменении базы данных или в случае, когда значение определенной переменной превысит заранее установленную границу.

QLISP является модификацией языка QA-4. Основное отличие заключается в том, что был исключен режим возвратов и вместо него введены такие же генераторы, как и в CONNIVER.

## 1.7. Система Smalltalk-80 и объектно-ориентированное программирование

Д. И. Безруков, А. О. Голосов

Недостатком традиционных языков программирования является наличие «семантического разрыва» между моделью предметной области и ее реализацией средствами языка. Соотнесение объекта и его языкового выражения (внешняя семантика модели) является серьезной проблемой, которую в значительной мере позволяет решить объектно-ориентированное программирование (ООП). Пробразом современных объектно-ориентированных языков программирования послужил известный язык моделирования Симула-67 [Дал и др., 1969]. В этом языке было введено понятие классов объектов и их иерархии. Многие идеи ООП, в частности идея описания поведения объектов в терминах операций, которые можно производить над объектами, заимствованы из абстрактных типов данных [Liskov et al., 1974; Liskov, 1976]. Первой наиболее успешной и перспективной разработкой в рамках ООП стали язык и система программирования Smalltalk-80 [Goldberg et al., 1983], получившая широкое распространение благодаря развитию ПЭВМ. Языки ООП наиболее успешно применяются в экспертных системах, при моделировании процессов с дискретными событиями, в системах автоматизации делопроизводства, базах данных и т. д.

При дальнейшем изложении основных концепций ООП будет применена терминология Smalltalk-80, которую использует большинство других систем ООП.

Основной конструкцией в ООП является *объект*. В качестве объектов могут выступать программистские абстракции (числа, символичные строки, файлы, редакторы текстов, программы, компиляторы и т. д.) или сущности моделируемой предметной области и их взаимосвязи (служащие, банковские счета, пациенты, врачи, диагнозы и т. д.). Объекты взаимодействуют друг с другом, посылая и принимая *сообщения*. Объект обладает собственной памятью для хранения информации и набором методов — операций для манипулирования этой информацией, называемых протоколом. Для активизации метода объекту посылается сообщение — аналог обращения к функции в традиционных языках программирования. В каждом сообщении должен быть указан адресат — объект, к которому посылается сообщение, имя сообщения (селектор) из протокола этого объекта и, возможно, несколько объектов-параметров, с которыми может манипулировать метод. Программа в системе ООП может быть представлена как последовательность сообщений к различным объектам.

Объекты объединяются в *классы*, причем каждый объект входит в один класс. В этом случае говорят, что он является *экземпляром класса*. Например, при моделировании учебного процесса в вузе каждый студент является экзем-

пляром класса Студент; экземплярами класса НатуральноеЧисло являются числа натурального ряда. Объекты из одного класса распознают один и те же сообщения и имеют одинаковую структуру собственной памяти — набор внутренних переменных объекта, называемых *переменными экземпляра*. Например, экземпляры класса ЦелыеЧисла распознают сообщения арифметических операций и операции сравнения; переменными экземпляра класса Студент являются: имяСт — имя студента, годРожд — год рождения, группа — номер учебной группы, в которой учится студент, и т. п. Для пояснения приведенных понятий можно провести следующую аналогию. Объект — это небольшая ЭВМ, переменные экземпляра — ее оперативная память, а набор распознаваемых сообщений — совокупность машинных инструкций.

У каждого класса существует два типа методов: методы классов, названия которых образуют протокол сообщений класса, и методы экземпляров, названия которых образуют протокол сообщений экземпляра. Методы классов используются, когда необходимо произвести какие-либо действия с целым классом, например добавить к классу новый экземпляр. Сообщения к экземплярам инициализируют методы, оперирующие с конкретным экземпляром данного класса.

Одна из ключевых идей ООП — построение иерархии классов (ISA-иерархии), поддерживающей механизм наследования свойств и тесно связанных с ним программный полиморфизм. Последнее обеспечивается специальными языковыми средствами, позволяющими создавать процедуры, одни и те же аргументы которых могут принимать значения различных типов (целые, текстовые, структуры и т. д.), существенно уменьшая сложность разработки больших программ.

Механизм наследования позволяет указать, что объекты одного класса наследуют свойства объектов другого класса (так организуется иерархия класс-подкласс). Подкласс содержит объекты, наследующие методы и переменные класса, называемого в этом случае *надклассом*, т. е. экземпляр подкласса наследует структуру собственной памяти экземпляра надкласса и распознает сообщения надкласса. Кроме того, структура его собственной памяти может включать новые имена переменных, а к протоколу сообщений экземпляра подкласса могут быть добавлены новые методы, удалены или модифицированы методы надкласса. Такой тип наследования используется в Smalltalk-80. В некоторых системах ООП, например Trellis/Owl [Schaffert et al., 1986], поддерживается более сложный механизм наследования — множественное наследование, при котором класс может иметь более одного надкласса.

Класс каждого объекта в системе обязательно является подклассом единственного «корневого» класса (В Smalltalk-80 он имеет имя Object), и все объекты могут наследовать сообщения этого «корневого» объекта, протокол которого содержит большое число методов, входящих в системную оболочку.

Описания каждого класса — переменных экземпляра и методов, в свою очередь, является объектом — экземпляром так называемого *метакласса*. Протоколы этих объектов содержат сообщения для создания и модификации классов.

В системе ООП существует ряд обязательных классов, обладающих примитивами, реализованными на языке низкого уровня и не подлежащими модификациям обычными средствами ООП. При выполнении любого сообщения в итоге обрабатывается один или несколько примитивов. Программист имеет возможность разработать и ввести в систему свои примитивы или заменить какой-либо метод примитивом (например, для повышения производительности).

Конкретная система ООП предоставляет программисту *системную оболочку* — базовый набор объектов и иерархий классов. Интерфейс пользователя, включая средства для создания объектов, запуск программ на счет, редактирование, копирование-восстановление файлов и другие операции, выполняемые обычно в среде операционной системы (ОС), в среде ООП реализуются в объектах системной оболочки. Объект, представляющий список, распознает сообщения об извлечении или помещении в список других объектов. Объекты — редакторы текстов — распознают сообщения о редактировании текстовых объектов (файлов) и т. д. Системная оболочка легко модифицируется и расширяется.

Классы и объекты, создаваемые программистом, могут включаться в системную оболочку.

Система Smalltalk-80 — процедурная система ООП, пользовательский интерфейс которой ориентирован на использование графического дисплея и устройства типа «мышь». Взаимодействие между объектами в Smalltalk-80 реализуется через сообщения. Посылка сообщения инициирует выполнение метода, который в ответ на сообщение, в свою очередь, возвращает объект. Имя посылаемого сообщения должно содержаться в протоколе адресата.

*Выражением* здесь называется последовательность символов, задающая объект, — *значение выражения*. Выражения могут быть четырех типов: литералы, переменные, сообщения и блоки.

Среди *литералов* (или литеральных констант) различают: числа — объекты, представляющие числовые значения; символы — объекты, представляющие символы в какой-либо системе кодирования (этому литералу должен предшествовать знак «\$»); строки символов — объекты, содержащие последовательности символов в кавычках; имена — объекты, представляющие имена в системе (имя распознается по предшествующему ему символу #, два одинаковых имени не допускаются); массивы — объекты, представляющие собой простые структуры данных с нумерованными элементами. Массив представляется в виде литералов других типов, разделенных пробелом и заключенных в круглые скобки, перед которыми стоит знак #, элементом массива также может быть массив.

Память объекта состоит из *переменных*. Каждая переменная указывает на объект и имеет имя или номер. Имя или номер могут представлять объект в выражении. Smalltalk-80 поддерживает следующие переменные:

- разделяемые переменные класса, доступные каждому объекту класса, в котором они определены, и всем объектам его подклассов;

- глобальные переменные, доступные всем объектам в системе;

- переменные, доступные объектам классов, явно указанных программистом.

Необходимость такой классификации связана с тем, что в практических приложениях не всегда объекты подкласса наследуют все свойства класса, например класс Птица, имеющий свойство летать, и его подкласс Пингвин, не обладающий этим свойством. Для помещения ссылки в переменную используется оператор присваивания. Несколько системных псевдопеременных, применение которых в левой части оператора присваивания запрещено, представляют специальные объекты: nil — ссылка на «пустой» объект, true и false — ссылки на объекты, представляющие «истину» и «ложь» соответственно, self и super используются в качестве адресатов при обращении из метода какого-либо объекта к самому себе.

*Сообщения* по форме задания подразделяются на три типа:

- унарные — содержат только адрес и селектор (имя сообщения из протокола адресата) без аргументов;

- бинарные — сообщения с одним аргументом, причем селектор представлен одним или двумя неалфавитно-цифровыми символами;

- сообщения с ключевыми словами — наиболее распространенный тип.

Селектор сообщения состоит из одного или более ключевых слов в фиксированной последовательности, причем после каждого ключевого слова задается двоеточие и объект-аргумент. Селектор сообщения определяется своими ключевыми словами, включая двоеточия. Например, в сообщении, создающем экземпляр класса Студент:

Студент зачислитьСтудентаИмя: фио  
годаРождения: 1967 вГруппу: 'Э25'

селектором является «зачислитьСтудентаИмя:годаРождения:вГруппу:»

Объект-значение возвращается всегда вне зависимости от того, пользуются ли им в дальнейшем. В качестве адресата и параметров в сообщении могут быть объекты, представленные возвращаемыми значениями сообщений. Выражения, отделенные друг от друга точкой, называются *предложениями*. Предложение выполняется слева направо, причем унарные сообщения более приоритетны, чем

бинарные, которые, в свою очередь, имеют более высокий приоритет, чем сообщения с ключевыми словами. Порядок выполнения сообщений можно менять, используя круглые скобки.

**Блоки-объекты** служат для задания последовательности действий и представляют собой набор предложений, заключенных в квадратные скобки. Когда блок встречается в программе, предложения, заключенные в скобки, не выполняются немедленно. Значение блока представляет собой объект, который может впоследствии выполнить заданные в нем выражения после получения унарного сообщения value. Через сообщения в протоколе объектов, представляющие различные структуры данных и содержащие блоки в качестве параметров или аргументов, реализуются такие языковые конструкции, как условный выбор (аналог if-then-else), условные повторы (аналог операторов while и until), циклы.

Большинство классов системной оболочки Smalltalk-80 — это различные структуры данных. Особый интерес представляет набор классов, обеспечивающий работу с объектами других классов, к которым относятся *коллекции* и *словари*. Протоколы различных коллекций позволяют обрабатывать совокупности объектов с неупорядоченными и упорядоченными элементами, могут допускать или не допускать дублирование элементов. Словари позволяют связывать пары произвольных объектов, запоминать и извлекать информацию о таких связях.

Управляющие структуры Smalltalk-80 обеспечивают не только традиционные логические возможности (циклы, if-then-else и т. д.), но и представление независимых процессов и механизмов для их синхронизации и взаимодействия.

Несколько классов системной оболочки предназначены для облегчения процесса программирования. В среду программирования входят классы для представления исходного и откомпилированного текста. Объекты, представляющие компиляторы и декомпиляторы, позволяют осуществлять преобразование текстов. Объекты, представляющие классы, связывают методы с экземплярами этих классов. Символьные отладчики также представлены объектами. Системная оболочка включает также объекты для просмотра и редактирования информации (включая тексты программ). Так как система ориентирована на растровые дисплеи, существует набор классов для манипулирования памятью экрана, включая выбор шрифтов для символов, представление прямоугольных окон, работу с меню и т. д.

Система Smalltalk-80 обеспечивает работу с внешними устройствами. Через соответствующий набор объектов предоставляются стандартные возможности файловой системы ЭВМ, а также работа в сети.

В качестве примера рассмотрим программу для решения известной головоломки «Ханойские башни». Имеется три штырька #1, #2, и #3, на штырьке #1 расположены кольца разного диаметра в порядке убывания их размера снизу вверх. Требуется переложить кольца со штырька #1 на штырек #2, пользуясь штырьком #3 как промежуточным, таким образом, чтобы кольца на штырьке #2 располагались в порядке убывания их диаметра и чтобы в процессе перекладывания кольцо большего диаметра никогда не накладывалось на кольцо меньшего диаметра. Ниже приведено описание метода «ханойС:на:через:», который следует поместить в описание класса целых чисел. Сообщение, инициирующее этот метод, адресуется к числу, задающему количество колец. В качестве аргументов выступают символы, задающие номера штырьков.

#### «ОПИСАНИЕ МЕТОДА»

```
ханойС: х на: у через: z
self < 1 ifTrue: [self error: 'неверен адресат'].
self ? 1 ifTrue:
    [Transcript nextPutAll 'Переложить со штырька', х,
                        'на штырек', у; cr]
    ifFalse: [
self — 1 ханойС: х на: z через: у.
Transcript nextPutAll 'Переложить со штырька', х,
                        'на штырек', у; cr.
```

self — 1 ханойС: z на: у через: x]

# «ВЫЗОВ МЕТОДА С ДЕСЯТЬЮ КОЛЬЦАМИ»:

10 ханойС: '1' на: '2' через: '3'

В результате выполнения метода в специальное окно Transcript на экране печатаются строки (сообщение `nextPutAll`), каждая из которых выдает команду о перекладывании кольца с одного штырька на другой.

В системе Smalltalk-80 выделяются два главных компонента: виртуальный образ и виртуальная машина. Виртуальный образ состоит из всех объектов в системе, виртуальная машина — из аппаратуры и программ на машинном языке, которые приводят в действие объекты из виртуального образа. Виртуальная машина имеет две основные части: интерпретатор и объектную память. Текст методов, используемый для ответов на сообщения, представлен в откомпилированном виде в форме однобайтовых инструкций, из байт-кодов. Интерпретатор выполняет байт-коды в виде последовательности инструкций рекурсивной стековой машины. Объектная память содержит представление структурных зависимостей между объектами.

Недостатками существующих в настоящее время реализаций являются низкая производительность и ограничение на общее число объектов и максимальный размер объекта. Кроме того, существенными ограничениями являются отсутствие традиционных средств, обычно предоставляемых системой управления базой данных (СУБД): мультидоступ к данным, механизмы сохранения и восстановления данных, обеспечение целостности и секретности данных. Важно и то, что языком Smalltalk-80 в силу его сложности, по-видимому, может воспользоваться только квалифицированный программист.

Требования систем с ООП к аппаратуре, на которой они должны работать, отличаются от всех распространенных в настоящее время компьютерных архитектур, и для их эффективной реализации, возможно, будут созданы специализированные ЭВМ. Простая, красивая, а главное, естественная для человека парадигма, предложенная в ООП для описания структуры и закономерностей предметной области, обеспечения полиморфизма, упрощающего написание очень сложных программ, оказалась чрезвычайно плодотворной при разработке прикладных интеллектуальных систем. В настоящее время исследования ООП проводятся по следующим основным направлениям:

создание объектно-ориентированных языков, как общецелевых, так и ориентированных на различные приложения (особенно в областях экспертных систем и автоматизации делопроизводства) [Ishiwkawa et al., 1986; Neirstrasz, 1985; Samples et al., 1986; Yonezawa et al., 1986; Yasuhiko et al., 1986];

дополнение существующих языков, особенно функционального и логического программирования, концепциями ООП [Moon, 1986; Bobrow et al., 1986; Fukunada et al., 1986];

создание объектно-ориентированных ОС, в том числе и распределенных [Anderson, 1986; Ewing, 1986];

разработка объектно-ориентированных СУБД [Maier et al., 1984; Maier et al., 1986].

### Языки и системы представления знаний

#### 2.1. Программные средства представления знаний: состояние исследований и проблемы

*В. Ф. Хорошевский*

Для настоящего этапа исследований в области представления знаний характерна концентрация усилий в следующих направлениях:

разработка систем представления знаний путем прямого использования широко распространенных языков обработки символьной информации и реже языков программирования общего назначения;

расширение базовых языков искусственного интеллекта до систем представления знаний за счет специализированных библиотек и пакетов;

создание языков представлений знаний, специально ориентированных на поддержку определенных формализмов, и реализация соответствующих трансляторов с этих языков.

Для всех основных языков символьной обработки имеются положительные примеры использования каждого из перечисленных выше направлений. В рамках каждого базового языка явным образом выделяются и прямое его использование, и расширение за счет пакетов функций, и создание «автономного» языка представления знаний (ЯПЗ) с последующей интерпретацией или компиляцией программ на созданном языке. Но в последнем случае базовый язык, как правило, становится инструментальным средством для реализации ЯПЗ. Таким образом, выделенные этапы составляют как бы один уровень жизненного цикла разработки систем представления знаний (СПЗ).

Независимо от реализации ЯПЗ должен отвечать следующим требованиям: наличие простых и вместе с тем достаточно мощных средств представления сложно структурированных и взаимосвязанных объектов;

возможность отображения описаний объектов на разные виды памяти ЭВМ;

наличие гибких средств управления выводом, учитывающих необходимость структурирования правил работы решателя;

«прозрачность» системных механизмов для программиста, т. е. возможность их доопределения и переопределения на уровне входного языка;

возможность эффективной реализации, как программной, так и аппаратной.

Конечно, перечисленные требования во многом противоречивы. Но лишь тогда, когда в рамках разумного компромисса учтены все эти требования, создаются удачные языки и системы представления знаний.

#### Системы представления знаний первого поколения

В 70-х годах в программировании вообще и программировании интеллектуальных задач в частности центр тяжести стал смещаться от процедурных к декларативным системам. К этому времени сформировались концепции представления знаний на основе семантических сетей и фреймов. Появились специальные языки программирования, ориентированные на поддержку этих концепций. Это был период бурного «языкотворчества» в области представления знаний. Но из языков представления знаний первого поколения лишь несколько сыграли заметную роль в программной поддержке СПЗ. Среди них, на наш взгляд, явно выделяются KRL, FRL, KL-ONE [Bobrow et al., 1977; Brachman et al., 1981, Goldstein et al., 1977; Niwa et al., 1984; Ангелова и др., 1984]. Характерными чертами этих языков были: двухуровневое представление данных (абстрактная модель предметной области в виде иерархии множеств понятий и конкретная модель ситуации как совокупность взаимосвязанных экземпляров этих понятий);

представление закономерностей предметной области и связей между понятиями в виде присоединенных процедур; семантический подход к сопоставлению образов и поиску по образцу.

Один из самых интересных языков этой группы KRL, в основу которого были положены следующие концепции: организация знаний в виде специально выделенных единиц с присоединенными описаниями и процедурами; наличие средств представления многоаспектного и/или неполного знания об объектах; возможность описания объектов через сопоставление их с другими объектами с учетом уточняющих описаний; наличие гибкого набора базовых средств описания стратегий вывода решений и возможность их динамического изменения. Однако KRL широко не использовался в интеллектуальных системах [Lehnert et al., 1979; Bohrow et. al., 1979a] из-за некоторой его громоздкости и отсутствия собственных средств описания процедур. Как следствие, в KRL активно использовался Лисп. Часто нельзя было понять, имеем ли мы дело с KRL-программой и присоединенными Лисп-функциями или с Лисп-программой, в которой применяется KRL как способ представления данных.

Язык FRL не самостоятельный язык, а хорошо продуманная библиотечная система над Лиспом. В FRL не предлагается принципиально новых по сравнению с KRL концепций представления знаний, но тем не менее он оказался более удобным благодаря тщательному и экономному отбору базовых алгоритмических средств, а также более простому их синтаксическому оформлению. Здесь имеются развитые средства манипулирования иерархическими списками свойств объектов, включая механизмы наследования свойств, и набор присоединенных к описаниям процедур. У нас в стране система программирования на базе FRL реализована в системе Лисп-МЭИ, а в последнее время и на ПЭВМ типа IBM PC/XT и выше с использованием muLisp [Байдун, 1980]. Язык KL-ONE имеет много общего с FRL как по уровню языка, так и по методам реализации. Но в отличие от последнего он базируется на концепции семантических сетей и предоставляет разработчику набор специальных средств для работы с ними.

Таким образом, к концу 70-х годов была сформирована значительная коллекция методов и языков представления знаний. Все это способствовало формированию новой стадии исследований в области искусственного интеллекта — переходу от экспериментальной программной проверки идей и методов к созданию практически значимых интеллектуальных систем. Однако переход от исследовательского программирования к промышленному требует модификации некоторых положений. Так, принцип «понятность программы важнее ее эффективности» уже не может больше служить основой разработки новых систем представления знаний. Переход к созданию прикладных интеллектуальных систем вызвал к жизни еще одну тенденцию — ускоренное развитие инструментальных средств поддержки разработки систем искусственного интеллекта. С точки зрения разработки и реализации систем представления знаний эти тенденции привели к появлению работ в области создания специализированных языков и систем программирования для представления знаний.

### **Разработка и реализация языков представления знаний**

Для всех ЯПЗ по сравнению с традиционными языками программирования характерна существенно большая «активность» данных, что приводит к стиранию граней между декларативной и процедурной компонентами. Кроме того, реальные объемы обрабатываемых данных требуют при реализации ЯПЗ использования концепции базы данных и методов, развитых при создании СУБД. И, наконец, как следствие первых двух обстоятельств, а также в силу существенного усложнения в ЯПЗ компоненты управления их реализацией тяготеет больше к режиму интерпретации, чем к компиляции, характерной для реализации обычных языков программирования. В области разработки и реализации ЯПЗ можно выделить три круга проблем: определение входных языков СПЗ; выбор выходного языка соответствующего транслятора и собственно проблемы этапа трансляции.



Инженер по знаниям, как правило, хочет иметь дело не с фиксированным ЯПЗ, а с семейством таких языков или, что еще лучше, иметь средства конструирования специализированных ЯПЗ, адекватных его задачам. Положение усложняется еще и тем, что входной язык СПЗ должен быть близок к языку предметной области и по лексике, и по синтаксису, и по семантике. В настоящее время концепция «Знания+Вывод=Система» находит свое отражение в ЯПЗ, где теперь явно специфицируются не только декларативная и процедурная компоненты (как в традиционных языках программирования), но и дифференциальная (inference — вывод) компонента. В любом языке программирования компонента управления выводом тоже присутствует, но она настолько проста, что специально об этом не говорится. Поток управления практически однозначно определяется последовательным выполнением операторов. В ЯПЗ-программах неоднозначность потока управления не исключение, а правило, и в каждый момент может быть применимо множество операторов. Поэтому требуются специальные соглашения о том, как будет организован этот процесс.

Огромную роль в реализации любого ЯПЗ играет выбор выходного языка, в конструкции которого транслируются исходные тексты. От этого зависит не только эффективность, но и сама возможность реализации ЯПЗ. Выходной язык должен отвечать по крайней мере следующим требованиям: иметь достаточно мощный набор примитивов работы с образцами; обладать встроенными средствами эффективной поддержки рекурсии; иметь гибкие средства описания потоков управления. Кроме того, в рамках выходного языка необходимы средства отображения данных на основную и внешнюю память и удобные средства работы с этими данными. И, наконец, желательно, чтобы в нем имелись достаточно развитые средства определения новых типов данных.

В настоящее время языков программирования, где имела бы место эффективная реализация всех указанных требований, пока нет. Поэтому выбор целевого языка ЯПЗ-транслятора всегда компромисс. Если в качестве выходного языка выбирается Лисп, то (как правило) приходится жертвовать требованиями отображения данных в базе данных и типизацией данных. И все же Лисп активно используется в качестве языка-подложки при реализации ЯПЗ. Это связано с наработкой огромных библиотек Лисп-функций, где реализованы и мощные средства работы с образцами, и гибкое управление, и многое другое. В языке Пролог изначально вопросы отображения данных на внешнюю память и поиск по образцу по внешней памяти были проработаны лучше, чем в Лиспе. Вместе с тем в Прологе достаточно жесткие ограничения на сложность образцов и простой (правда, встроенный) механизм вывода решений. Естественно поэтому, что в настоящее время все чаще предпринимаются попытки объединить достоинства Пролога и Лиспа в рамках единого языка программирования, который был бы удобен и эффективен для решения интеллектуальных задач.

С точки зрения разработчиков трансляторов, ЯПЗ являются новым этапом в этой области. По существу, здесь мы переходим от «классических» методов синтаксически управляемой трансляции к СПЗ о процессах реализации языков программирования. А это, в свою очередь, приводит к созданию специализированных экспертных систем (ЭС), в рамках которых знания разработчика входного ЯПЗ в режиме диалога интегрировались бы со знаниями в таких областях, как анализ и синтез текстов (в том числе и естественных языковых), методы трансляции, синтез программ и т. п.

### Современные языки представления знаний

В настоящее время языков и систем представления знаний сотни, и количество их все увеличивается. Поэтому ниже приведены лишь некоторые замечания относительно трех наиболее интересных, на наш взгляд, ЯПЗ: RLL (Representation Language Language), ART (Automated Reasoning Tool), OPS5 (Official Production System, version 5) [Greiner et al., 1980; ART, 1984; Brownston et al., 1985]. Первый из них RLL выбран как представитель достаточно популярного в 70-х годах подхода «фреймы до конца» и как пример удачного инстру-

ментального подхода к проблеме автоматизации разработки ЭС. Язык ART демонстрирует другую парадигму «фреймы плюс продукции», характерную для начала 80-х годов, а OPS5, претендующий на роль языка-стандарта в области представления знаний для ЭС, показывает дальнейшее смещение акцентов в сторону продукции, закрепленное в концепции «продукции плюс фреймы». Вместе с тем в настоящее время уже редко удается классифицировать языки и системы представления знаний на простой шкале «фреймы — продукции — семантические сети — ...» однозначно. И хотя тот или иной формализм представления знаний накладывает в большей или меньшей степени свой отпечаток на соответствующий ЯПЗ, современные языки и системы, как правило, поддерживают несколько формализмов одновременно. Фреймовый подход активно используется в системах медицинской диагностики, таких, как MEDAS, NEUROLOGIST, PIP и др. [Ben-Bassat et al., 1980; Szolovits et al., 1979], а также в базе медицинских знаний [Артемьева и др., 1983]. Здесь разработано несколько версий ЯПЗ МЕДИФОР (МЕдицинский ДИАгностический ФОРмализм) [Клещев и др., 1978; Горбачев, 1984]. В последней версии этого языка (МЕДИФОР-3) достаточно хорошо проработаны вопросы описания декларативных и (в меньшей степени) процедурных знаний, но дифференциальная компонента почти полностью «спрятана» на уровне трансляции текстов МЕДИФОР-программ и в ядре программного обеспечения базового языка РЕЛЯП [Артемьева и др., 1984].

RLL. Это фреймовый язык представления знаний, является инструментальной средой для создания специализированных ЯПЗ.

Подобно другим инструментальным средствам, RLL содержит два слоя: базисные примитивы и средства их комбинирования на более высоком, чем Лисп, уровне. При этом технология конструирования специальных ЯПЗ в рамках RLL-среды сводится, как правило, к редактированию уже существующих заготовок и последующему конвертированию их в Лисп.

Учитывая последовательную ориентацию RLL на концепцию фреймов, все структуры (декларативные и процедурные), более сложные, чем список значений, описываются здесь в виде фрейм-подобных RLL-элементов:

M6-3	
Isa	Люк
К-нему:	M6-2
Расположен-под:	(Улица-А Ориентир-1)
M6-2	
Isa	Люк
От-него:	M6-3
Расположен-под:	(Улица-А Улица-6)
Люк	
Isa:	Set
Generalization:	(Физ-объект, ...)
Examples:	(M6-2, M6-3, ...)
Description:	Так описываются все люки
Prototype:	Типовой люк
New-Slots:	(К-нему, От-него, Расположен-под)
К-нему	
Isa:	Slot
Description:	Втекающий поток
Inverse:	От-него
To-compile:	(Lambda (m) (OneOf (Remove m (...))))

С помощью RLL-элементов описываются понятия не только предметной области, но и самой RLL-среды (например, слот, механизм исследования, структура управления и т. д., и т. п.). Заранее на уровне RLL-интерпретатора или конвертора фиксируется семантика ограниченного числа системных понятий — это множества, списки, слоты и др. RLL-элементы имеют явно специфицирован-

ные родовидовые отношения, которые также являются системными понятиями, и встроенный механизм описания отношений с помощью многосвязных списков. В рамках RLL используется аппарат присоединенных процедур, которые могут использоваться как для получения значений тех или иных слотов, так и в качестве демонов, включающихся в работу при использовании понятий, где они описаны.

Правила обработки тоже являются RLL-элементами, в которых имеются специальные if- и then-слоты. Однако в силу инструментального характера RLL-среды здесь должны быть более гибкие и общие средства спецификации процессов обработки RLL-элементов. Например, в описании RLL-правила

#### Правило #332

Isa	Rule
Description:	При работе с токсичными веществами необходимо применять специальные меры
Если-Потенциально-Релевантно:	(химическая токсичность высокая?)
Если-Действительно-Релевантно:	(локализация (около человека)?)
Тогда-Скажи:	«Данное вещество не вдыхать!»
Тогда-Добавить-к-Agenda:	(список подцелей)
Приоритет:	Высокий
Стоимость:	900
Время-Прогона:	0.1 с
Частота-Использования:	4 раза/985 случаев
Generalization:	(Правило #899, Правило #45)
Specialization:	(Правило #336)
Автор:	Джонсон
Дата-Создания:	17:30 9 июля 1981 года

несколько if- и then-слотов. Такое разделение позволяет связать с каждым из типов слотов свой механизм вывода решений, а также установить приоритетность различных механизмов и тем самым повысить эффективность обработки RLL-объектов. Так, в приведенном выше правиле механизм вывода, связанный со слотом «Если-Потенциально-Релевантен», выбирает множество потенциально подходящих правил быстро (в силу простоты проверки описанного в этом слоте условия), а другой механизм, например, выбирает из этого множества лишь те правила, время работы которых не превышает определенного порога. Возможны и другие стратегии вывода решения, например всегда следовать правилам одного и того же автора.

В RLL имеется и библиотека удачных управляющих структур, и определенные средства конструирования из них решателей, необходимых для конкретной ЭС.

Одним из основных стандартных механизмов вывода решений в RLL является agenda (дословный перевод «повестка дня» не вполне отражает семантику этого механизма, которую точнее было бы идентифицировать как «управляющий список с динамической коррекцией элементов»). Agenda-процессор в RLL-среде реализует практически те же алгоритмы, что и процессор обработки RLL-правил, который, в свою очередь, аналогичен процессору обработки RLL-элементов и слотов. Учитывая это, стандартный подход к созданию специализированных управляющих процессоров в RLL-среде — копирование процессора правил, его редактирование с целью получения процессора задач и, наконец, повторное редактирование для получения agenda-процессора.

ART. Это не только язык представления знаний, но и определенное программное окружение, включающее редакторы, отладчики, трансляторы и модули управления.

Входной язык системы ART весьма гибкий и обеспечивает использование фактов, схем, комбинаций этих понятий и правил. Декларативную компоненту этого ЯПЗ составляют факты и схемы. По определению, факт включает три основных компонента: утверждение, значение истинности и точку зрения. С каждым утверждением может быть связано одно из трех значений истинности true, false или unknown, а также определенные сферы его справедливости (контексты истинности), которое и называется точкой зрения. Факты описываются экземплярами фреймов. Фреймы-прототипы в ART представляются схемами, каждая из которых описывает объекты и/или классы объектов с фиксированными свойствами. Механизмы наследования свойств при этом поддерживаются самой системой.

В ART понятие точки зрения употребляется не только как контекст истинности факта, но и в более широком смысле, чтобы ввести в рассмотрение возможные «состояния мира». Этот же механизм используется для представления процессов, разворачивающихся во времени, а также для того, чтобы обеспечить псевдопараллельное применение всех допустимых на каждом шаге вывода решения продукционных правил, составляющих процедурную компоненту языка ART.

В общем случае в базе правил системы ART могут храниться правила трех типов: гипотетические, ограниченный и полаганий. Все они имеют традиционную для продукций «если-то» форму, а отличаются способами работы с контекстами.

При использовании правил первого типа, общую структуру которых можно описать словосочетанием «если нечто случилось, можно предполагать что-то», формируются гипотетические миры (точки зрения, контексты). На следующих шагах вывода решения эти миры существуют независимо друг от друга и «развиваются» до тех пор, пока появление некоторой информации в базе фактов не приведет к тупиковому выводу в некотором из миров. При появлении таких ситуаций соответствующие миры уничтожаются автоматически.

Правила ограничений описывают ситуации, которые не должны появляться в «хороших» контекстах. Поэтому их семантика представляется следующим образом: «если нечто имеет место, данный гипотетический мир нужно уничтожить». Таким образом, если представлять структуру контекстов как динамическое дерево, гипотетические правила будут использоваться для генерации новых его листьев, а правила ограничений — для свертки тех ветвей, которые не нужны для получения решения.

Несколько сложнее в ART правила полаганий. В первом приближении их семантику можно описать следующим образом: если достигнута некоторая цель, контекст, в котором это произошло, «хороший». Кроме того, «подтверждающий» контекст соединяется по определенным правилам со всеми своими (и только с ними) предшественниками и в результате создается новый корневой контекст в структуре вывода решений. В целом этот механизм похож на управление в языке Плэнер, но в ART автоматически проверяются все «осиротевшие» контексты на совместимость их с новым корнем. Таким образом, с помощью правил полагания в ART можно в динамике уменьшить размерность дерева поиска решений и осуществлять определенные перестройки дерева.

В целом язык ART погружен в Лисп-среду, так что синтаксически и фреймвые, и продукционные структуры выражаются здесь как атомы, списки и функции языка Лисп. Такой подход в ART естествен, так как первоначально был реализован на Лисп-машинах. Средства описания фактов в языке ART почти полностью «отданы на откуп» Лиспу, что снижает концептуальную целостность языка, так как средства описания схем и правил здесь хотя и похожи на лисповские, но свои. Что же касается инференциальной компоненты, то в ART принята следующая концепция: пользователю дается небольшой набор встроенных стратегий вывода решений и весьма ограниченный выбор из ART-действий, взаимодействующих с модулем вывода. Но в системе имеется возможность выхода в базовый язык Лисп, где программируются любые управляющие стратегии.

Для иллюстрации возможностей языка ART ниже приводится фрагмент программы для решения известной модельной задачи о волке, козе и капусте: Определение контекста:

(Def-Viewpoint-Levels (merging nil) state)

Определение схем:

```
(DefSchema Человек «для местоположения человека»  
  (Location Берег-1))  
(DefSchema Обладание-чем-то «содержит схемы...»  
  (Has-Instances Волк Коза Капуста))  
(DefSchema Волк «первоначально на Берег-1»  
  (Instance-Of Обладание)  
  (Location Берег-1))
```

Определение фактов:

```
(DefFacts Противоположность «Берега противоположны»  
  (Opposite-Of Берег-1 Берег-2)  
  (Opposite-Of Берег-2 Берег-1))
```

Определение правил:

```
(DefRule Движение «перемещение человека»  
  (Schema Человек)  
  (Location Берег-2))
```

→

```
(Sprout state  
  (Modify Человек  
    (Location Берег-1)))
```

```
(DefConstraint Волк-ест-козу «не оставлять волка с козой»  
  (Schema Волк  
    (Location ?Берег))  
  (Schema Коза  
    (Location ?Берег))  
  (Opposite-Of ?Берег ?Другой-Берег)  
  (Schema Человек  
    (Location ?Другой-Берег)))
```

В полном объеме ART предоставляет разработчику ЭС достаточно мощные средства представления знаний, но эффективно в системе ART могут работать только квалифицированные Лисп-программисты, готовые реализовывать в этом языке все процедуры поддержки ЯПЗ.

**OPS5.** Это один из ЯПЗ, основанных на производственной модели представления знаний. В общем случае OPS5-программа содержит секцию деклараций, где описываются используемые в программе объекты и определенные пользователем функции, и секцию производных, включающую правила. Во время исполнения обрабатываемые данные помещаются в рабочую память, а правила — в производную память. Обычно рабочая память инициализируется после декларации данных и загрузки правил.

Модуль вывода решений в OPS5 состоит из трех основных блоков: отождествления, где осуществляется поиск подходящих правил; выбора исполняемого правила из множества подходящих; исполнения выбранного правила. Непосредственно в OPS5 поддерживается единственная стратегия вывода решения — вывод, управляемый целями, с отождествлением по элементам правил из рабочей памяти, выбором единственного исполняемого правила и последовательным выполнением действий из его правой части. Вместе с тем OPS5 — достаточно гибкий язык программирования, в котором имеются явные средства не только для описания данных, но и для определения потоков управления над этими данными. Правда, это требует достаточно высокой квалификации инженера по знаниям.

Описание OPS5-объектов в конечном счете сводится к определению экземпляров фреймов, прототипы которых в OPS5-программе задаются в виде определенных структур данных, построенных на основе небольшого числа встроенных типов данных. В языке OPS5 используется всего два примитивных скалярных типа: числа (целые и действительные) и символьные атомы. Для более сложных типов данных OPS5 следует к языку Паскаль с его развитым аппа-

## Алгоритмические средства описания объектов в OPS5

OPS5-действие или функция	Пояснения
(make имя-класса имя-1 значение-1 имя-2 значение-2 ..... имя-N значение-N)	Позволяет создать объект заданного класса с заданным списком свойств и их значений
(remove ссылка-1 ссылка-2 ..... ссылка-N)	Позволяет удалить перечисленные в качестве аргументов объекты
(modif имя-класса имя-1 значение-1 имя-2 значение-2 ..... имя-N значение-N)	Позволяет изменить значения определенных свойств определенного объекта
(bind переменная [значение])	Используется при необходимости запоминания промежуточных значений на уровне одного правила
(build структура)	Позволяет в динамике генерировать объекты и правила

ратом определения новых типов данных, а Лиспу, который является языком реализации OPS5. Конечно, наличие в ЯПЗ развитых механизмов определения новых типов данных весьма желательно. Поэтому развитие языков OPS-семейства, по-видимому, пойдет в этом направлении. Пример такого подхода — OPS83, где переменные типизированы более жестко, чем в Паскале [Brownston et al., 1985]. Но даже в OPS5 некоторые средства определения составных структур данных уже имеются. В первую очередь, это так называемые элементы класса с компонентами-атрибутами и векторы атрибутов, с помощью которых формируются описания фреймов-прототипов (правда, очень простых). Элементы рабочей памяти, получаемые из этих прототипов, строятся как списки свойств. При этом имена атрибутов маркируются префиксным оператором ↑. Ниже для примера приводится описание элемента рабочей памяти с именем «Личность»:

(Личность

↑ ФИО	Иванов И. И.
↑ Мать	Петрова
↑ Возраст	10
↑ Адрес	. . . . .
. . .	. . . . .)

Для создания элементов рабочей памяти с уровня OPS5 существует несколько способов. Один из них — использование команды make:

```
(Make Город XXXXXXXXX
      Название YYYYYYYY
      Расположение ZZZZZZZZ)
      Страна
```

Для этого имеются и другие возможности, некоторые из них представлены в табл. 2.1.

Интересно в OPS5 использование так называемых временных тегов, идентифицирующих такты, на которых элемент рабочей памяти был определен и/или модифицирован. Эта информация анализируется модулем вывода решений при разрешениях конфликтного множества.

Как и в других продукционных языках, OPS5-правила идентифицируются уникальными именами и имеют левую (совокупность условных элементов) и правую (последовательность действий) части. Для OPS5 характерно то, что условные элементы являются здесь образцами, отождествление которых осуществляется через поиск подходящих элементов в рабочей памяти. Правила в OPS5 могут загружаться в продукционную память до начала работы специальными Р-командами. Но правила могут и генерироваться динамически из правой части некоторой продукции с помощью действия build.

Каждый условный элемент в левой части продукции является, по существу, спецификацией ограничений на некоторый элемент рабочей памяти, а вся левая часть — множеством условных элементов и/или их отрицаний, описывающих вместе ситуацию применимости данного правила. Учитывая вышесказанное, нетрудно представить алгоритмические средства задания образцов в OPS5 (табл. 2.2). Вместе с тем наследование свойств, например, уже не выражается этими простейшими средствами и требует в OPS5 гораздо более изощренного программирования.

Правая часть OPS5-правила — последовательность действий, каждое из которых не что иное, как Лисп-функция, встроенная в OPS5-интерпретатор и/или определения на уровне базового языка (Лиспа) инженером по знаниям. Основных действий в OPS5 всего 12. Кроме них в языке определены и несколько стандартных встроенных функций, связанных с арифметикой, генерацией новых символьных атомов, форматной печатью. Ниже приводятся примеры OPS5-правил, иллюстрирующие основные конструкции этого ЯПЗ:

; ЕЯ-запись продукционных правил:

```
; IF проверяются входные данные и значение возраста,
; заданное пользователем, больше 130 лет
; THEN выдать сообщение об ошибке и удалить данное значение
;
```

```
(Р проверка-возраста : неверное-значение : слишком-большое
 (context goal проверка-входных-данных)
 {(input token {input-value>130}) input})
```

```
→
 (write (crlf) Величина input-value превышает 130)
 (remove input))
```

```
;
; IF проверяются входные данные и значение возраста,
; заданное пользователем не число
; THEN выдать сообщение об ошибке и удалить данное значение
;
```

```
(Р проверка-возраста : неверное-значение : не-число
 (context goal проверка-входных-данных)
 {(input token {input-value $\neq$ nil}) input})
```

```
→
 (write (crlf) Величина input-value не число)
 (remove input))
```

## Средства задания образцов в OPS5

Условный элемент (образец)	Пояснения
(имя-класса)	Образец отождествляется с любым элементом рабочей памяти того же класса
(имя класса ↑ имя-1 значение-1 ↑ имя-2 значение-2 ..... ↑ имя-N значение-N)	Образец отождествляется с теми элементами рабочей памяти того же класса, которые имеют те же атрибуты, что и у образца, плюс (быть может) дополнительные атрибуты
(имя-класса ..... ↑ имя <<значение-1 значение-2 ..... значение-N>> .....)	Образец с дизъюнкцией значений атрибута, значения задаются литерально
(имя-класса ..... ↑ имя { значение-1 значение-2 ..... значение-N} .....)	Образец с конъюнкцией значений атрибута, значения могут быть и литералами, и условными выражениями

В OPS5 цикл работы модуля вывода включает три стадии: отождествление, выбор и исполнение. Их взаимодействие можно описать следующей спецификацией:

repeat

    выполнить «отождествление»;

    выполнить «выбор правила из конфликтного множества»;

    выполнить «исполнение выбранного правила»

until

    («конфликтное множество не пусто») или

    («не выполнено действие halt») или

    («счетчик числа циклов не достиг заданной величины») или

    («не достигнута контрольная точка»)

Таким образом, модуль вывода решений базируется на последовательном выполнении трех достаточно сложных процедур. Процедура выбора подходящего правила из конфликтного множества поддерживает две встроенные стратегии выбора подходящей продукции LEX и MEA. LEX-стратегия может быть описана следующим образом:

begin LEX-стратегия

    begin стадия-рефракции

        выполнить «удаление из конфликтного множества всех правил, которые выполнялись на предыдущем шаге»;



```

end стадия-рефракции;
begin стадия-лексикографического-упорядочивания
  выполнить «сортировку по убыванию значений временных
    тегов элементов рабочей памяти»;
  выполнить «удаление правил с временными тегами, значения которых
    меньше заданного порога»;
  if (к-во правил в конфликтном множестве=1)
    then exit
  else begin
    выполнить «сортировку правил по числу условных элементов
      в них и по факторам важности внутри каждой
      группы, удаляя менее «важные» правила»;
    if (к-во правил в конфликтном множестве=1)
      then exit;
    end;
  end стадия-лексикографического-упорядочивания;
begin стадия-детализации
  выполнить «сортировку по убыванию к-ва условий в правилах»;
  выполнить «удаление правил, к-во условий в которых меньше заданного
    порога»;
  if (к-во правил в конфликтном множестве=1)
    then exit
  else выполнить «случайный выбор правила»;
  end стадия-детализации;
end LEX-стратегия

```

Отличие MEA-стратегии состоит в том, что сразу после стадии рефракции здесь включается стадия более детального исследования первого условного элемента каждого из оставшихся правил с целью упорядочивания их в конфликтном множестве по критерию цель-подцель. Если этого сделать не удастся, выбор подходящего правила осуществляется в соответствии с LEX-стратегией.

Язык OPS5 — достаточно мощный продукционный инструмент. Вместе с тем с точки зрения базового набора операций уровень OPS5 ниже, чем у обобщающегося ЯПЗ первого поколения, уступает он по мощности примитивов и языку RLL.

## 2.2. Язык представления знаний оболочки СПЭИС

О. В. Ковригин

В области представления знаний практически одновременно возникли и развивались два направления: продукционные системы и представление с помощью семантических сетей (фреймов). Развитие ЯПЗ базировалось в основном на продукционных системах. С появлением более сложных продукционных систем современные гибридные средства проектирования ЭС базируются преимущественно на продукционных и объектно-ориентированных системах, которые позволяют манипулировать сложными иерархически организованными объектами в рамках продукционных систем.

### Представление знаний

В системе СПЭИС используются два формализма представления: объектно-ориентированный и продукционный. *Продукционная часть* языка предназначена для моделирования стратегий рассуждений эксперта (метазнания) и представления проблемных знаний в виде правил. *Объектная часть* языка служит для описания базовых понятий проблемной области, их иерархии, а также для представления понятий, используемых для реализации моделей рассуждения.

Основные примитивы ЯПЗ, который предлагается пользователю системы оболочки СПЭИС, также описываются в виде объектов, что позволяет относительно просто перестраивать базовый набор элементов ЯПЗ высокого уровня.

**Объектно-ориентированная часть ЯПЗ.** Каждое понятие в СПЭИС может быть определено как объект со списком свойств. Значением свойства может быть число, строка, атом, список, множество функций (процедур) и продукций. Каждый объект может принадлежать некоторому классу. Класс (в свою очередь, объект) используется для описания свойств принадлежащих ему объектов. Описание класса состоит из трех независимых частей: описание методов получения значений для каждого свойства; описание набора действий, производимых после означивания свойства; описание схемы наследования свойств и методов. Следуя уже сложившейся терминологии, первую группу функций называют слугами, а вторую — демонами.

Слуги и демоны свойств могут определяться с внутренними переменными: self (сам объект), pror (само свойство) и value (значение свойства). Они сохраняются в объекте после выполнения функций, при следующем обращении к свойству, при запуске слуг и демонов начальные значения внутренних переменных связываются с запомненными значениями. Обращение к слугам и демонам возможно с аргументами, которые связываются с переменными, описанными в разделе (var:) тела слуги или демона.

Приведем пример определения класса объектов, представляющих элементы некоторой технической системы:

```
(Defclass Element Servants:                                ;определение «слуг»
  Inputs: (ask «Inputs of» self)
  Outputs: (ask «Outputs of» self)
  Status: (GetKnow Status Value: self)
  Demons:                                           ;определение «демонов»
    Inputs: (addprop value Outputs: self)
    Outputs: (addprop value Inputs: self)
    Inher: all)                                     ;наследуются все свойства
(Defclass Status Servants:
  Value: ((var: x)
    (ask «Enter status of element» x)))
```

При определении класса Element динамически строится функция с именем Element. При применении этой функции ее аргументы, т. е. объекты, относящиеся к данному классу, помещаются в свойство Examples: данного класса, а в каждый объект (экземпляр) данного класса под свойство Is-a помещается имя класса.

После описания классов определяются, например, два элемента системы, связанные между собой (E1→E2) и относящиеся к классу Element:

```
(Element E1)
(Element E2)
```

В результате в свойство Examples: объекта (класса) Element будет помещен список (E1 E2), а в объекты E1 и E2 под свойство Is-a — атом Element. Определив элемент E1 как объект со списком свойств:

```
(DefObject E1 Outputs: E2)
```

получим следующую реакцию системы: заполнив слот Outputs: значением E2, система обратится к описанию класса (Element) с целью отыскания соответствующей функции-демона. Если такая функция существует и может быть наследована согласно информации в слоте Inher:, то она выполняется. В нашем примере это приведет к занесению в свойство Inputs: объекта E2 атома E1. Это пример организации рекурсивных свойств объектов.

Если потребуется определить значение свойства Status: для элемента E1, достаточно обратиться с запросом (Getp E1 Status:). В этом случае система рассмотрит элемент E1 и его свойство Status:. Если оно неизвестно, то

система перейдет на один уровень иерархии вверх по свойству Is-a (если это позволяет описанная в Element схема наследования) и попытается «достать» значение этого слота для объекта Element. Если и эта попытка окончится неудачей, то система перейдет еще на один уровень иерархии. В результате если найдется объект в родовом дереве, имеющий означенное свойство Status, то это свойство будет наследовано объектом E1. Иначе система попытается отыскать метод означивания свойства, также двигаясь по иерархии свойства Is-a. В нашем случае такой метод будет найден в объекте Element, в разделе описания класса (свойство Servants:). Там находится функция, предписывающая послать запрос объекту Status с целью определения значения его свойства Value:.. Как определить это значение, описано в разделе Servants: описания класса Status. Обратим внимание на передачу аргумента слугой свойства Status: объекта Element слуге свойства Value: объекта Status (х будет связано со значением self, т. е. E1).

Таким образом, объектно-ориентированная часть языка в СПЭИС позволяет описывать классы объектов, не налагая строгих ограничений на значения свойств, организовать наследование означенных свойств по иерархии, а также демонов и слуг. Один объект может принадлежать нескольким классам одновременно и наследовать разные свойства из разных классов. Одна и та же структура может одновременно выступать как объект с собственным набором означенных свойств и как описатель класса.

**Продукционная часть ЯПЗ.** Часть базы знаний, содержащая описания объектов, является основой для построения и исполнения продукций. Продукции представляют собой правила вида

If <условие> Then <действие> ... <действие>

Левая часть правила <условие> — это логическая комбинация предикатов. В условии допускается использование операций AND, OR и NOT. Различаются предикаты, использующие: переменные, например

(If (Color=red) ...или (If (Color & red blue green)...

где на первом месте стоит имя глобальной переменной, на втором — предикативный символ (операции =, /, >, <, >=, <= и операции над множествами: = (/=) — равенство (неравенство) с точностью до перестановки элементов множества, операция пересечения множеств (^), операция включения множеств (&)), на третьем месте — значение (атом, список или число), или другая глобальная переменная, или вычисляемая функция;

простые списки свойств, например

(If (Color of Bus=Color of Car)...) )

где на первом месте стоит обращение к значению свойства некоторого атома, на втором — предикативный символ (тот же, что и в первом случае) и на третьем — атом, число, список, обращение к свойству атома либо вычисляемая функция;

сложный механизм настройки на образ, например

(If (Diagnose > name ^ object < disease ^ cf (> 0.7))...

это условие будет истинным, если в базе данных найдется хотя бы один объект класса Diagnose (имя этого объекта будет связано с переменной name), такой, что у него имеется свойство object, значение которого должно совпадать со значением ранее связанной переменной disease, и свойство cf, значение которого должно удовлетворять ограничению (больше 0.7). Результатом настройки правил последнего вида на образ может быть множество объектов класса Diagnose. Для выбора одной подстановки используются специальные стратегии разрешения конфликтов.

Правые части правил содержат действия, которые могут включать: означивание глобальных переменных и свойств объектов; удаление, добавление и изменение свойств объектов;

динамическое создание и удаление классов и экземпляров объектов; любые функции языка Лисп.

Таким образом, смешанный ЯПЗ в системе СПЭИС базируется на трех компонентах: объектах, правилах, функциях (процедурах), что позволяет описывать знания различного рода и создавать как ЭС, основанные на простых правилах, так и приложения, требующие наличия причинно-следственных и имитационных моделей в базах знаний.

### Модификация формализма представления знаний и вывода решений

В системе-оболочке СПЭИС разработчику ЭС предоставляется возможность создать свою собственную модификацию ЯПЗ и программу вывода решений. Для этого используются элементы объектно-ориентированных языков программирования, которые позволяют определить независимые классы объектов, а также разрешают постепенное и независимое наращивание системы и облегчают модификацию программного комплекса.

Класс также может рассматриваться как объект, являющийся примером некоторого другого класса. Это позволяет реализовать механизм наследования свойств для определения значений атрибутов объектов. Объект может относиться сразу к нескольким классам и наследовать свойства из каждого класса. Информация о том, какие свойства объект должен наследовать из каждого класса, может быть задана явным образом при определении класса. Описание метода означивания свойств необходимо проводить на языке Лисп. При этом можно использовать примитивы формализма представления знаний СПЭИС.

Приведем пример некоторых понятий формализма представления, принятого в системе-оболочке пакета СПЭИС. Так, в ЯПЗ системы-оболочки определены два понятия: Р-концепция и Н-концепция. Каждая конкретная Р-концепция является примером понятия Р-КОНЦЕПЦИЯ, а класс Р-КОНЦЕПЦИЯ — примером более общего понятия ОБЪЕКТ. Для класса ОБЪЕКТ определены основные элементарные операции, например заполнения и извлечения значений атрибутов. Описание класса ОБЪЕКТ может быть следующим:

```
(defclass Object put ((var: x y z)
                      (putprop x y z))
  get ((var: x y)
       (getprop x y))...)
```

Теперь можно сказать, что Р-КОНЦЕПЦИЯ — это объект (Object Р-КОНЦЕПЦИЯ), и определить Р-концепцию как некоторый класс:

```
(defclass Р-КОНЦЕПЦИЯ
  get: ;метод как обрабатывать запрос GET:
  ((var: par)
   (message self infer-value par) ;попробовать вывести значение
   (message self ask-value par) ;спросить значение
   (message self association par)) ;активизировать кластер

  name ((var: par) ;метод как достать и напечатать имя
        (message screen window par) ;окно для печати имени
        (message screen col-attr red blue) ;цветовые атрибуты
        (message screen print
          (message par get name))) ;достать имя параметра

  infer-value ;метод как выводить значение признака
  ((var: par)
   (message Inference-Engine par ;анализ продукции
    (message par get depends))) ;продукции для вывода

  ask ;метод как задавать пользователю вопросы
  ((var: par) ;и получать ответы
   (if (exists (message par get value)) ;если известно
```

```

then stop      ;то не спрашивать
else           ;иначе
  (message self name par)    ;требование печатать имя
  (message screen window)    ;возможные значения в окно
    (message par get possible-value)
    (message screen col-attr cyan black)
    (message screen print
      (message par get possible-value)
      (message Selection answer par))

```

Теперь каждую конкретную Р-концепцию (пусть это будет Р4) можно определять как пример класса Р-КОНЦЕПЦИЯ и далее конкретизировать свойства концепции Р4:

```

(DS P4
  (IS-A . Р-КОНЦЕПЦИЯ)
  (PNAME САМЫЕ ВЫСОКИЕ ЦИФРЫ ДИАСТОЛИЧЕСКОГО
    АРТЕРИАЛЬНОГО ДАВЛЕНИЯ)

  (POSSIBLE-VALUE . number)

  (RESTRICTION (between 0 280))

  (DESCRIPTION P6 P16)

  (FRESTRITION (less-then P3))

  (STR-CONTROL
    (1 ((IF (AND (P3 <= (P5) (P4 <= (P6)))) (THEN (ACTIVATE: END)
      (STOP))))))

```

Представление концепции Р4 на языке, близком к естественному:  
Концепция Р4

имя  
САМЫЕ ВЫСОКИЕ ЦИФРЫ ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО  
ДАВЛЕНИЯ

возможные значения ЧИСЛО

ограничения МЕЖДУ 0 280

ограничения! МЕНЬШЕ ЧЕМ САМЫЕ ВЫСОКИЕ ЦИФРЫ  
СИСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ

ассоциативная связь

ВОЗРАСТНАЯ НОРМА ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО  
ДАВЛЕНИЯ

ЦИФРЫ ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ РАНЬШЕ  
управление выводом

1 ЕСЛИ

И

САМЫЕ ВЫСОКИЕ ЦИФРЫ СИСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО  
ДАВЛЕНИЯ < =

ВОЗРАСТНАЯ НОРМА СИСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО  
ДАВЛЕНИЯ

САМЫЕ ВЫСОКИЕ ЦИФРЫ ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО  
ДАВЛЕНИЯ < =

ВОЗРАСТНАЯ НОРМА ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО  
ДАВЛЕНИЯ

ТО  
АКТИВИЗИРОВАТЬ Н-КОНЦЕПЦИЮ: ДАВЛЕНИЕ В ПРЕДЕЛАХ  
НОРМЫ  
СТОП:

Разработчик ЯПЗ может определить свои новые понятия, используя уже имеющиеся, например примитивы класса ОБЪЕКТ или понятия Inference-Engine (в последнем реализованы методы работы с продукционными системами).

Таким образом, инженер по знаниям может модифицировать ЯПЗ системы-оболочки и построить собственный ЯПЗ, отвечающий его требованиям и традициям.

### 2.3. Язык представления знаний Пилот

*В. Ф. Хоросhevский, С. Ю. Щенников*

Язык Пилот относится к ЯПЗ продукционно-фреймового типа. Это значит, что ПИЛОТ-программу следует рассматривать как набор продукционных правил, обрабатывающих некоторое множество элементов фреймоподобной структуры. Из сказанного естественно вытекает синтаксическое разбиение Пилот-программ на две части: элементную и продукционную:

```
start имя-программы
      элементная_часть
      продукционная_часть
```

В элементной части задаются прототипы (схемы) фреймоподобных элементов (фактов). Прототип задает внутреннюю логическую структуру факта, которая в общем случае имеет вид

```
(имя-факта
 {имя-слота [=значение-слота]
 {имя-слота [=значение-слота]}})
```

Каждый прототип описывается с помощью конструкции #prototype. В ней указываются имя прототипа, имена его слотов и их возможные значения. Возможным значением слота может быть либо литерная строка, либо ссылка на другой прототип. Ссылка идентифицируется по указанию имени соответствующего прототипа на месте значения слота.

Продукционная часть Пилот-программы состоит из набора продукционных правил (продукций). Весь набор продукций может быть разбит на отдельные поименованные секции. Каждую секцию можно рассматривать как подсистему системы всех продукций Пилот-программы. Синтаксически этот компонент описывается следующим образом:

```
продукционная_часть
 {секция-продукций} +
 секция-продукций
 section имя-секции
 {секционное-разрешение}
 {продукция} +
 продукция
 rule имя-продукции
 {правилное-разрешение}
 условие
 действие
```

В языке Пилот нет встроенной стратегии разрешения конфликтного множества. Поэтому Пилот-интерпретатор, обрабатывая иницированную секцию продукций, случайным образом выбирает из конфликтного множества какое-

либо ее правило для применения. Однако часто известны эвристические правила усечения и упорядочивания конфликтного множества. Для описания таких правил предназначены конструкции «секционное-разрешение» и «правилоразрешение». Синтаксически оба вида разрешений предполагают наличие в них правил вида посылка: следствие. Вместе они предназначены для описания определенной стратегии усечения конфликтного множества, применяемой в данной секции продуктов. Предполагается, что стратегия усечения может зависеть от одного из четырех факторов или их комбинации: полный состав конфликтного множества; наличие в конфликтном множестве указываемых продуктов; применение известных продуктов; наличие в базе фактов тех или иных элементов и определенные значения переменных. Отличие секционного разрешения от правилоразрешения состоит лишь в том, что в посылках первого нельзя использовать элемент «вхождения в конфликтное множество», а в посылках второго — элемент «полный состав конфликтного множества». Для выражения каждого из перечисленных факторов предназначены следующие виды элементов посылок правил разрешения:

полный-состав-конфликтного-множества

ifall имя-продукции { , имя-продукции }

Данная конструкция интерпретируется следующим образом: «если конфликтное множество состоит только из тех продуктов, имена которых указаны после ifall, то ...»;

вхождение-в-конфликтное-множество

ifwith имя-продукции { , имя-продукции }

Результат обработки этого элемента Пилот-интерпретатором полностью зависит от того, внутри какого продукционного правила он находится: «если в конфликтном множестве вместе с данным продукционным правилом присутствуют также правила, имена которых указаны после ifwith, то ...», в отличие от конструкции ifall ifwith-элемент не требует полного совпадения своего аргумента с конфликтным множеством:

использование-продукционных-правил

ifused имя-продукции { , имя-продукции }

В процессе выполнения программы Пилот-интерпретатор заполняет специальный системный стек именами использованных продуктов. Данная конструкция позволяет описать условие на упорядоченное вхождение в стек задаваемых имен продуктов: «если указанные после ifused продукционные правила были ранее применены в порядке их перечисления, то ...».

Выбор из конфликтного множества предпочтительных продуктов можно поставить в зависимость от содержимого базы фактов и текущих значений переменных. Для этого в посылке правил разрешения целесообразно использовать элемент «состав базы фактов и переменные», начинающийся с ключевого слова ifrat. После слова ifrat в общем случае может стоять список образцов и их отрицаний, соединенных логическими связками. Структура такого списка полностью совпадает с конструкцией условной части продукционного правила и обрабатывается Пилот-интерпретатором аналогично. Если в посылке правил разрешения несколько элементов, то они считаются связанными конъюнкцией.

Следствие правила разрешения может содержать действия одного из двух типов: удаление указываемых продукционных правил из конфликтного множества и коррекция его содержимого. В последнем случае просто перечисляются имена тех продуктов, которые следует оставить в конфликтном множестве:

следствие

|| remove имя\_продукции { , имя\_продукции } ||  
имя\_продукции { , имя\_продукции } ||

В конструкции remove может быть не указано ни одного имени продукта. Это соответствует удалению из конфликтного множества того продукционного правила, в котором находится данная конструкция.

Обратим внимание на три важных момента: 1) правила разрешений начинают анализироваться только тогда, когда в конфликтном множестве находится хотя бы одно производственное правило; 2) все правила разрешения с удовлетворенными посылками выполняются только один раз в порядке их следования в тексте Пилот-программы; 3) из всех правил разрешения для возможного использования анализируются правила из конструкции секционного разрешения и правила из тех разрешений, чьи производственные правила попали в конфликтное множество.

*Условная часть производственного правила* представляет собой набор образцов и их отрицаний. В общем случае они могут быть соединены между собой логическими связями И (& или пусто) и ИЛИ (∨);

условие

condition  $\left\| \begin{array}{l} \text{образец } \{ \vee \text{ образец} \} \\ \text{образец } \{ \& \text{ образец} \} \\ \text{образец } \{ \text{образец} \} \\ \text{образец} \end{array} \right\|$

образец

$\left\| \begin{array}{l} \text{—простой-образец} \\ \text{—(образец)} \\ \text{простой-образец} \end{array} \right\|$

На уровне простого образца допускается использование двух типов конструкций, различных по синтаксису и семантике. К первому типу относятся простые образцы, которые при своей интерпретации только сопоставляются с элементами базы фактов (БФ), назовем их *БФ-образцами*. Ко второму типу относятся образцы, в которых описывается сравнение двух объектов, строк или фактов, будем называть их *С-образцами*.

Логическая структура факта представляет собой упакованную запись вида «имя факта — имя слота — значение слота». Эта идея была распространена и на синтаксическую структуру БФ-образца, за тем лишь исключением, что здесь используются не компоненты факта, а их шаблоны:

шаблон

$\left\| \begin{array}{l} * \\ \text{П-переменная} \\ \text{составная-строка} \end{array} \right\|$

Шаблон \* сопоставим с любым компонентом. Любому компоненту сопоставима и П-переменная, однако в этом случае она «захватывает» в качестве своего нового значения копию сопоставленного компонента, т. е. происходит означивание переменной. В языке Пилот поддерживаются глобальные переменные. Это означает, что их значения доступны в любой части Пилот-программы в любой момент времени. Все переменные делятся по видам их обработки на два класса: П-переменные (захватывают значение); О-переменные (отдают свои значения). Переменные имеют стековую организацию с дисциплиной обслуживания LIFO, поэтому для перемещения по уровням стека необходимы соответствующие операторные средства. В языке принято, что работать можно только со значением на самом верхнем уровне стека переменной (в соответствии с дисциплиной обработки элементов стека) или, точнее, с последним захваченным ею значением. Изменить содержимое переменной можно, либо удаляя (замещая) верхний уровень стека, либо образуя новый уровень с новым значением (возможно, пустым). Для этого введены четыре оператора префиксной формы — по два для каждого класса переменных. Для П-переменных «>» — образовать новый уровень и занести на него захваченное значение и «>>» — заменить текущее значение верхнего уровня новым захваченным значением. Для О-переменных «<» — скопировать содержимое верхнего уровня стека и «<<» — скопировать значение верхнего уровня и спуститься на уровень



ниже, удалив при этом бывший верхний. Очевидно, что *O*-переменная типа «<» никак не изменяет содержимое своего стека.

Шаблон «составная\_строка» является вычисляемым, заменяется конкретной (литерной) строкой, и лишь после этого происходит собственно сопоставление (посимвольное сравнение) полученной строки с соответствующей компонентой выбранного факта. Оно считается успешным, если литерная строка и компонента полностью совпали. Составная строка рассматривается как конкатенация подстрока. В качестве знака конкатенации используется символ «.» (точка).

подстрока

пустая_подстрока литерная_подстрока <i>O</i> -переменная наследование арифметическое_выражение
--

Семантика пустой строки интуитивно ясна. *Литерная строка* — это неделимая цепочка символов, заключенная в апострофы. Конструкция *наследование* является вычисляемой, ее интерпретация соответствует выполнению некоторого абстрактного параметризованного оператора «взять значение слота факта». В качестве параметров оператора принимаются имя факта и имя слота, указываемые в рассматриваемой конструкции. Результатом вычисления будет либо литерная строка, идентичная значению слота указанного факта, либо пусто, если в базе данных не было найдено факта с заданным именем или указанного слота у данного факта. Значение будет пустым и в том случае, когда слот факта не имеет значения. Имена фактов и слотов в конструкции «наследование» не всегда являются в точности литерными строками, а представляют собой в общем случае составные строки. Такая конструкция обладает большой гибкостью, позволяя, в частности, описывать косвенное указание. Арифметическое выражение предназначено для описания вычислений, которые необходимо провести. Описываемая здесь версия языка Пилот поддерживает лишь целочисленную арифметику.

Проиллюстрируем вышесказанное примером БФ-образца. Для этого используем следующий фрагмент Пилот-программы:

```

section S1
  rule r11
    condition (вершина12: Тип='Целевая')
    action ...
  rule r12

```

Смысловое содержание условной части правила r11 следующее: «если в базе фактов есть факт с именем «вершина12», имеющим слот «Тип» со значением, идентичным строке «Целевая», то...».

На начальном этапе разработки языка С-образец предполагался как описание обычного сравнения двух литерных строк на лексикографический порядок. В дальнейшем С-образец был развит до сравнения двух объектов: либо (как прежде) строк символов, либо фактов из базы фактов. При этом наиболее интересным объектом сравнения являются факты. Они идентифицируются собственными квадратными скобками, внутри которых помещается шаблон имени факта. Неформальная интерпретация знаков сравнения (в рамках конструкции С-образца с объектами-фактами) следующая. Два факта называются идентичными, если они имеют один и тот же набор слотов с одинаковыми значениями. Говорят, что факт Ф1 включает факт Ф2, если Ф1 имеет в своем множестве слотов в виде подмножества полный набор слотов факта Ф2. В любых других случаях факты неидентичны и один факт не включает другой. Для выражения в С-образце рассмотренных понятий используются знаки «=», «>»,

«<» и их комбинации. Будем говорить, что факты сравнимы, если произошло успешное сопоставление фактов для заданного знака сравнения.

Включение в язык конструкции «сравнение» объектов-фактов обусловлено недостаточной мощностью выразительных средств БФ-образца, который, например, не позволяет сопоставить непосредственно два факта из базы фактов. Может показаться, что выход из такого положения в последовательном сопоставлении двух данных фактов с одним образцом. Но это справедливо, лишь когда состав сравниваемых фактов (с точностью до значений слотов) точно известен и, следовательно, можно описать соответствующий БФ-образец. Однако возможны ситуации, когда структура и состав фактов заранее неизвестны, но провести их сравнение необходимо.

Возьмем для примера динамический механизм вывода, работающий по стратегии forward-chaining (прямой вывод) по схеме «сначала виришь». Суть работы данного механизма заключается в последовательной генерации всех новых возможных гипотез из ранее полученных по определенным эвристическим правилам. Процесс генерации заканчивается, если гипотеза эквивалентна цели, заданной до начала процесса вывода, либо если далее не может быть сгенерировано ни одной новой гипотезы. Для такого механизма вывода естественно потребовать не генерировать гипотезы, полученные на предыдущих шагах вывода. Представив гипотезы как отдельные факты с уникальными именами, можно использовать конструкцию «сравнение» объектов-фактов, указав имя факта, представляющего текущую обрабатываемую гипотезу. Таким образом, мы всегда будем фиксировать ситуацию повторной генерации ранее полученной гипотезы.

Положительным свойством данной конструкции является и то, что с ее помощью можно строить продукционные правила с динамически корректируемой условной частью. Для этого в базе фактов выделяется отдельный факт для описания посылки некоторой продукции. Естественно, при этом надо помнить о соответствии имен данного продукционного правила и выделенного факта-посылки. Используя в условной части продукционного правила С-образец с указанием имени факта-посылки, можно подменить такую условную часть содержанием факта-посылки.

Часть продукции «действие» представляет собой последовательность операций. Все операции в языке Пилот делятся на два класса — основные и неосновные:

действие

action { || основная-операция ||  
          || неосновная-операция || }

*Основной* является операция коррекции базы фактов, которую будем называть *манипулятором*. Манипулятор предназначен для описания изменений, которые требуется произвести в базе фактов. По своей синтаксической структуре он близок к БФ-образцу. Однако в отличие от последнего здесь можно задавать изменение компонента факта с помощью оператора замены «\». В зависимости от местонахождения оператора замены в конкретном манипуляторе он описывает удаление (добавление) факта из (в) базу(у) фактов либо изменение состава фактов. В первом случае оператор замены находится на месте имени факта, во втором — на месте слота. Оператор замены задает удаление факта (слота), если справа от наклонной черты используется конструкция «пусто». Если же конструкция «пусто» слева от черты, то это означает добавление факта (слота). Интерпретация частей оператора замены не изменяется, если его аргументы — непустые конструкции: левая из них определяет, что надо удалить, а правая — что добавить. Если же оба аргумента представлены конструкцией «пусто», такой оператор замены считается пустым и игнорируется Пилот-интерпретатором.

Приведем пример конкретного манипулятора:

section S

. . . . .

rule R  
condition ...  
action

(\* : Значение='1', Статус \ Статус='Рассмотрена')

Здесь используется ранее упомянутая Пилот-программа для механизма вывода, реализующая прямой вывод по схеме «сначала-вширь». Рассмотрим случай, когда в процессе вывода требуется перевести все подтвержденные вершины, т. е. имеющие слот 'Значение', равный '1', И-графа в разряд рассмотренных. Это означает, что в соответствующих фактах нужно изменить значение слота «Статус» на «Рассмотрена».

Использование в манипуляторе шаблонов имен фактов и слотов, значений слотов в сочетании с множественным сопоставлением образца манипулятора позволяет легко обходить ограничение единичности сопоставления для БФ-образца с помощью пассивных манипуляторов. В соответствии со своим названием он не производит изменений в базе фактов (в нем отсутствуют непустые операторы замены), и Пилот-интерпретатор осуществляет только первый этап его выполнения. Последнее позволяет, не меняя базу фактов, получить имена фактов, сопоставленных неявному образцу данного манипулятора.

*Неосновные операции* условно разбиты на пять групп: управления, управления активностью продукции, обмена с внешним носителем, интерфейса с пользователем и прямого присваивания.

1. Операции управления: запуск, возврат и выход. *Операция запуска* иницирует выполнение некоторого программного модуля. Синтаксически — это вызов функций с параметрами. В качестве имени программного модуля могут быть использованы имена секций продукции, продукционных правил, Рефал-программ (Рефал-функций) и С-программ (С-функций). При запуске секции Пилот-интерпретатор переходит на обработку запускаемой секции, запоминая адрес точки возврата. При запуске продукции безусловно выполняются операции из их правых частей. При запуске Рефал-программ и С-программ в иницируемых программах необходимо учесть соглашения о связях между Пилот-программами и программами, написанными на языках Рефал и Си. Рассматриваемая версия языка Пилот не поддерживает конструкции операции запуска, в которой имя программного модуля есть в общем случае вычисляемая составная строка. В более поздних версиях языка реализация этого свойства планируется на базе динамического вызова исполняемых программ.

*Операция возврата* (return) предназначена для возврата Пилот-интерпретатора на обработку секции продукции, которая была ранее иницирована, но еще не закончила свою работу. Разрешено два вида возврата: возврат в правило и в секцию. При первом выполняются операции, оставшиеся в правой части продукционного правила, при втором эти операции игнорируются. В обоих случаях Пилот-интерпретатор продолжает обработку той секции продукции, на которую он был возвращен. Основным видом возврата является первый — с продолжением выполнения операций соответствующего продукционного правила. Для возврата второго вида необходимо задать символ «s» в качестве второго параметра операции возврата, в первом параметре указывается имя секции продукции, куда следует вернуться. Так как в языке Пилот разрешен вложенный вызов секций, интерпретатор работает не с одной парой адресов возврата, а со стеком адресов возврата, «поднимаюсь» по нему при запуске какой-либо секции (с занесением в стек соответствующей пары адресов) и «спускаюсь» при возврате в какую-либо секцию (с удалением из стека пары адресов). Данный стек фактически хранит цепочку вызовов секций продукционных правил. Задав определенный адрес возврата (имя секции), мы можем заставить интерпретатор «спуститься» сразу на несколько уровней в стеке адресов возврата, при этом пропущенные уровни будут удалены. Если стек адресов возврата в этот момент оказался пустым, то осуществляется выход из Пилот-программы в программный модуль, из которого она была вызвана, с сохранением содержимого базы фактов в оперативной памяти.

*Операция выхода* (exit) позволяет либо аварийно завершить выполнение Пилот-программы с выходом в операционную систему (возможно, с кодом возврата), либо нормально завершить выполнение Пилот-программы и вернуть управление в тот программный модуль, из которого она была запущена. В последнем случае можно заказать выход с сохранением или без сохранения базы фактов в оперативной памяти. Операция выхода без параметра осуществляет выход в операционную систему с нулевым кодом возврата.

2. Операции управления активностью продукций. Обрабатывая некоторую секцию, Пилот-интерпретатор проверяет удовлетворение посылок всех продукционных правил, входящих в нее. На практике встречаются случаи, когда требуется временно исключить из рассмотрения определенные продукционные правила или, наоборот, включить ранее исключенные правила. Это можно запрограммировать с помощью операций активизации *act* и деактивизации *deact* продукционных правил. В качестве их параметров указываются имена продукционных правил. При деактивизации продукционных правил Пилот-интерпретатор помечает их как правила с всегда неудовлетворенными посылками, при активизации правил такая отметка (если она была поставлена) снимается.

Примером интенсивного использования этих операций является Пилот-программа решения задачи о ханойской башне. Приведем ее фрагмент, в котором показана реализация переключения активности:

```

section SI
  rule R11
    condition
    action
      deact(R11)
      act(R12)
      .
      .
      .
  rule R12
    condition
    action
      deact(R12)
      act(R11)
      .
      .
      .

```

В данном фрагменте описана секция SI, состоящая из продукционных правил R11 и R12. Оба правила имеют пустые посылки, т. е. они всегда потенциально применимы. Правые части правил среди прочих содержат операции управления активностью продукционных правил: правило R11 деактивизирует продукцию с именем R11 (само себя) и активизирует R12. Правило R12 аналогично правилу R11. Выполнение любых конкретных операций управления активностью продукционных правил не влияет на последовательность выполнения других операций из правой части применяемого продукционного правила. Таким образом, на каждом цикле обработки секции SI, исключая, быть может, самый первый, в данной программе для Пилот-интерпретатора будет находиться только одно активное продукционное правило: либо R11, либо R12.

3. Операции обмена с внешними носителями: открытие-закрытие файлов, обмен для строк символов и обмен для базы фактов. Одновременно в разных режимах может быть открыто произвольное число файлов. Для открытия файла необходимо в параметрах операции *open* указать соответственно полное имя файла и режим работы с ним: чтение, запись и добавление. Для закрытия файла достаточно указать его имя в параметре операции *close*.

По существу в языке ПИЛОТ поддерживаются два типа данных — строки символов и факты. Для файловой работы с данными первого типа предназначены операции *put* и *get* — записать/прочитать в/из файл соответственно. Обе операции имеют по два параметра. В первом указывается имя файла, во втором для операции записи используется конструкция «составная строка» (ее численное значение записывается в файл), а для операции чтения ставится ли-

бо  $P$ —переменная, либо имена факта и слота, в которые переносятся прочитанная из файла строка символов.

Операции обмена для базы фактов чаще всего рассматриваются как операции загрузки-выгрузки базы фактов с внешнего носителя и на него. Операции этой группы имеют по два основных параметра. В первом, как обычно, указывается имя файла, второй представляет собой БФ-образец и введен для целей выборочной загрузки-выгрузки фактов. В описании обращения к операции выгрузки может присутствовать дополнительный параметр — флаг удаления, так как «по умолчанию» при выгрузке на внешний носитель очередной факта он не удаляется из базы фактов. Выставляя флаг удаления, мы тем самым заказываем удаление всех фактов, сопоставленных с заданным БФ-образцом, после их выгрузки на внешний носитель.

4. Операции интерфейса с пользователем: операции, работающие с экраном в обычном (телетайпном) режиме, и операции, работающие в многооконной среде в графике. Операции обычного режима — вывод на экран сообщения, чтение строки с экрана, выдача на экран содержимого фактов, сопоставленных с заданным образцом. Операция `out` выводит на экран сообщение — литерную строку, полученную в результате вычисления составной строки, которая в общем случае может быть параметром данной операции. Операция же также выводит сообщение на экран, но прерывает исполнение Пилот-программы, ожидая ответа пользователя. Его ответ воспринимается как строка символов и запоминается либо как значение  $P$ —переменной, либо как новое значение указанного слота заданного факта. При определенных параметрах выполнение операций, эквивалентно выводу сообщения на экран с последующим ожиданием нажатия любой клавиши. При выполнении операции `outfb` Пилот-интерпретатор ищет в базе фактов те факты, которые сопоставляются БФ-образцу из параметра операции, а затем выводит содержимое найденных фактов на экран дисплея в следующем формате:

имя\_факта:

имя\_слота\_1 = значение\_слота\_1

имя\_слота\_2 = значение\_слота\_2

имя\_слота\_N = значение\_слота\_N

В оконном режиме операции аналогичные, отличия их только в том, что выдача информации осуществляется в определенные окна. В данную подгруппу входят также операции, необходимые для создания многооконной графической среды. Синтаксически операции интерфейса оконного режима отличаются от аналогичных операций обычного режима лишь указанием дополнительных параметров: номера окна, с которым связывается данная операция, и цвета символов сообщения. Операции интерфейса графического многооконного режима реализованы на базе пакета `PEN`, разработанного в ВЦ АН СССР и по своим функциональным возможностям аналогичного пакету `HALO`. При выполнении операции определения окна Пилот-интерпретатор устанавливает для данного окна вертикальный ролинг, границы которого задаются так, чтобы в окне можно было поместить максимальное количество строк. В начале выполнения любой из выше перечисленных операций, на экране появляется окно, обрамленное рамкой, в левом углу — имя окна, а в правом — его номер.

5. Операция прямого присваивания. В первых версиях языка Пилот предполагалось, что означивание переменных может производиться при обработке соответствующих фактов. Однако практическая работа с языком показала необходимость средств непосредственного (прямого) изменения стека значений переменных. Поэтому была введена операция прямого присваивания вида

$P$ —переменная := составная\_строка

которая заносит в стек указанной  $P$ —переменной новое значение и выполняется в рамках соглашений, принятых в языке относительно работы с уровнями значений стека переменной. Присваиваемое значение является вычисляемым вы-

ражением. Способ работы с  $P$ -переменной, указываемой в левой части операции присваивания, полностью определяется видом ее префиксной части. Приведем примеры нескольких операций прямого присваивания:

```
section SI
rule RI
condition ...
action
```

```
>p1='вершина'
>p2=<p1.<number + '1'
>>p3 = <p2. '='. [<<p2 : 'Статус']
```

В активной части продукции  $RI$  выделены три операции прямого присваивания. Первая «заносят» строку символов 'вершина' на новый уровень стека переменной  $p1$ . Вторая операция также «заносят» на новый уровень стека переменной  $p2$  другую строку, получаемую в результате вычисления составной строки — конкатенации копии переменной  $p1$  и арифметической суммы копии переменной  $number$  и 1 (допустим, сумма оказалась равной 12). В результате новым значением (на верхнем уровне стека) переменной  $p2$  станет строка 'вершина12'. Третья операция — замена содержимого текущего уровня переменной  $p3$  новым значением, которое получается в результате конкатенации трех подстрок составной строки: копии переменной  $p2$  ('вершина12'), литерной строки '=' и значения слота 'Статус' факта, имя которого есть значение переменной  $p2$ . Если принять, что значение слота 'Статус' факта 'вершина12' есть 'Рассмотрена', то новым значением переменной  $p3$  будет строка 'вершина12= Рассмотрена'.

В настоящее время реализован транслятор с этого языка для ПЭВМ типа IBM PC и многооконный отладчик.

## 2.4. Язык представления лингвистических знаний ATNL-2.0

*В. Ф. Хорошевский*

В настоящее время разработаны и используются системы, ориентированные на общение с конечным пользователем на естественном языке (ЕЯ) [ТИИЭР, 1986]. Вместе с тем реализация лингвистических (Л) процессоров, поддерживающих такое общение, остается сложным и трудоемким делом. По-видимому, к решению такой задачи существует два подхода.

В рамках первого реализация Л-процессора осуществляется с помощью одного языка программирования или спектра подходящих языков. При этом сложность и трудоемкость реализации зависит от удачного выбора инструментальных средств, а эффективность реализации — от квалификации программиста и качества штатного программного обеспечения.

Второй подход предполагает создание специального языка для программирования Л-процессоров, который выступает в качестве языка представления лингвистических знаний. При этом предполагается, что эффективный транслятор с выбранного языка на данной ЭВМ уже реализован. Таким образом, при втором подходе центральное место занимает система представления лингвистических знаний. Такая система должна удовлетворять ряду требований, основным из которых являются простота и естественность, достаточная мощность языка представления лингвистических знаний, а также возможность эффективной реализации его на заданном классе ЭВМ или типе аппаратуры.

### Сетевой подход к проектированию Л-процессора

Один из широко используемых в настоящее время формализмов построения ЕЯ-систем — расширенные сети переходов (ATN), предложенные Вудсом [Woods, 1970], которые являются естественным развитием матриц переходов

Замельсона — Бауэра [Samelson et al., 1960] и диаграмм переходов Конвея [Conway, 1963]. Если метод Замельсона — Бауэра позволяет анализировать А-языки, а метод Конвея — КС-языки, то ATN-формализма Вудса — широкий класс моделей языков, включая естественные.

Во всех этих методах используется описание реализуемого языка в виде графа. Последовательное увеличение мощности достигается следующим образом. В формализме Замельсона — Бауэра все узлы графа помечаются нетерминальными символами, а единственным условием прохождения любой дуги является текстуальное совпадение терминала, которым она помечена, с очередным сканируемым символом. Семантическая обработка осуществляется ограниченным набором немедленно исполняемых команд. В формализме Конвея каждая из множества диаграмм переходов определяет некоторый синтаксический тип (нетерминал). Дуги диаграмм помечаются терминальными и нетерминальными символами. Условием прохода по дуге является либо текстуальное совпадение очередного сканируемого символа с терминальной пометкой дуги, либо успешное «вычисление» той диаграммы, именем которой помечена соответствующая дуга. Так организуется рекурсивное обращение к другим диаграммам. Для организации семантической обработки дуг помечаются действиями (подпрограммами), которые выполняются после завершения «вычисления» соответствующей диаграммы.

В целом ATN-формализм Вудса является расширением формализма Конвея за счет вынесения контекста в специальные именованные стековые регистры, введения специальных процедур управления ходом анализа на основе эффективной проверки контекстных условий, достаточно широкого спектра семантических процедур, моментом запуска и порядком выполнения которых управляет разработчик анализатора, более последовательного использования механизма возвратов (backtracking) и более гибкой дисциплины перебора возможных путей анализа. Вудс укрепил ATN-формализм в специальном метаязыке описания расширенных сетей переходов. Описание сети фрагментируется по кустам, каждый из которых задается своей вершиной и исходящими из нее дугами. Вудс различает четыре основных типа дуг TST, CAT, PUSH и POP. Последний тип дуг вводится для индикации заключительных состояний в сети.

Для прохождения любой из дуг требуется истинность Р- и С-условий. Р-условия определяются как композиция И—ИЛИ—НЕ—РАВНО предикатов, а также предикатов Т и NIL, аргументами которых являются литеральные константы и(или) значения форм. С-условия различны для различных типов дуг и поддерживаются встроенным механизмом. С-условия для TST- и POP-дуг истинны по умолчанию, для CAT-дуг истинны, если текущее слово входного текста совпадает с литеральной пометкой этой CAT-дуги или имеется в словаре, имя которого указано на данной CAT-дуге. На PUSH-дуге С-условие задается именем состояния, а вычисление значения истинности связано с рекурсивным обращением к определенному этим именем подграфу и возможностью выхода из него через одно из заключительных состояний. В случае неудачи на подграфе срабатывает встроенный механизм возвратов и С-условие становится ложным, движение по PUSH-дуге прекращается и вычисляется истинность С- и Р-условий на следующей дуге куста.

Проход по дуге сопровождается, как правило, выполнением действий, связанных с записью значений в именованные регистры, работающие как LIFO-стек. Формирование этих значений и их считывание из регистров осуществляется в рамках ATN-формализма с помощью небольшого базового набора форм. Для перехода к следующему кусту используется переход без считывания очередного слова входного текста или со считыванием его в специальный системный регистр.

В настоящее время существуют различные варианты языков описания расширенных сетей переходов [Bols, 1983]. Однако основное внимание в этих языках уделяется описанию сети. Вводятся дополнительные типы дуг, расширяется состав предикатов, действий, форм и т. п. Однако только средств описания сети для практического использования ATN-формализма недостаточно. Нужны раз-

витые средства описания декларативной компоненты знаний (например, средства описания словарей), а также более гибкие средства управления (каскадные ATN [Woods, 1980], управляемые возвраты [Кох и др., 1981]).

### Язык ATNL-2.0

Теоретической базой языка представления лингвистических знаний ATNL-2.0 являются расширенные сети переходов Вудса. В СССР разработаны и реализованы несколько версий языка ATNL [Хорошевский, 1979]. Однако, как показала практика, для эффективного применения его проблемными специалистами требуется существенное упрощение нотации программ; для обеспечения последовательного накопления и проверки отдельных фрагментов конкретного Л-процессора и последующего объединения их в общий блок общения необходимо введение в язык простых и естественных средств модульности; для использования словарей реальной степени сложности и объема необходимо использование баз данных, а следовательно, и соответствующих средств описания в самом языке ATNL. Для повышения описательной мощности и упрощения реализации потребовалась модификация существующей версии языка. Полученный в результате такой модификации язык ATNL-2.0 и представлен в данном параграфе [Khoroshevsky, 1983].

В общем язык ATNL-2.0 предназначен для автоматизации проектирования Л-процессоров. Все знания, необходимые для реализации конкретного модуля ЕЯ-общения, представляются в виде ATNL-программы, которая с точки зрения проблемного специалиста разбивается на несколько разделов:

MODUL-DIVISION.	описание-раздела-модульности
VOCAB-DIVISION.	описание-раздела-словарей
[DEFINE-DIVISION.	описание-раздела-нестандартных-функций]
NET-DIVISION.	описание-раздела-сетей

раздел модульности: фиксация справочной информации о разрабатываемом Л-процессоре, описание точек объединения отдельных фрагментов Л-процессора путем декларации внешних по отношению к данному модулю объектов, определение тех точек в ATNL-программе, которые являются входными в данном модуле, описание синонимов внешних и внутренних объектов;

раздел словарей: описания структуры лексических единиц, используемых Л-процессором, и словарных статей, рекомендации по отображению словарей на различные виды памяти;

раздел нестандартных функций: определения композиций стандартных средств языка ATNL, повышающих наглядность программ, и описания алгоритмических средств, не определенных в рамках данного языка и/или повышающих эффективность работы Л-процессора;

раздел сетей: определение используемой в разрабатываемом Л-процессоре модели общения и описание сети анализа-синтеза фраз языка общения.

Каждый из разделов в соответствии с описываемыми типами знаний может быть разбит на секции, а секции, в свою очередь, на параграфы и/или фразы. Так, раздел модульности содержит две секции: *секцию идентификации*, в которой фиксируется в виде отдельных фраз имя модуля, информация о разработчике и дате проектирования, а также назначение модуля; *секцию связи*, где дается описание синонимов, внешних объектов и входных точек. Для иллюстрации конструкций, используемых в разделе модульности, предположим, что имеется два фрагмента Л-процессора, первый разработан для анализа именных групп, а второй — для анализа предикатов. Объединяются эти фрагменты с помощью мониторов, который обеспечивает анализ запросов, содержащих некоторые именные группы и предикаты. Тогда в каждом из трех модулей Л-процессора появляется свой раздел модульности. Один из них, мониторинг, представлен ниже:

MODUL-DIVISION.  
IDMOD-SECTION.  
MODNAME-IS                    МОНИТОР



AUTHER-IS	РУКОВОДИТЕЛЬ
DATE-IS	10.09.82.
COMMENT-IS	ЭТО ГОЛОВНОЙ МОДУЛЬ Л-ПРОЦЕССОРА.
LINKAGE-SECTION.	
ИМ-ГР, АГЕНТ, МЕСТО, ОБЪЕКТ, ИНСТР	ARE-EQV.
R-ИМ-ГР, R1, R-ГЛ-ГР, R2	ARE-EQV.
ИМ-ГР, R-ИМ-ГР, R-ГЛ-ГР	ARE-EXTRN.
ПРЕДЛОГ	WITH-TYPE VOC ARE-EXTRY.
ПРЕДИКАТ	WITH-TYPE LABEL IS-ENTRY.
ЗАПРОС	IS-ENTRY.
РЗАПР	

Из приведенного примера могут быть легко выделены форматы фраз раздела модульности языка ATNL-2.0.

Раздел словарей также содержит две секции. В первой — *секции шаблонов* — вводится структура морфологической модели языка общения. Средства ее описания должны быть естественными с точки зрения интуиции разработчика и вместе с тем достаточно мощными и гибкими для описания различных морфологических моделей с требуемой степенью детализации. Первая секция раздела словарей содержит последовательность описаний шаблонов словарных единиц, каждый из которых определяется следующим форматом:

имя-шаблона = элемент-шаблона {+ элемент-шаблона};

где

элемент-шаблона ::= NIL | имя-словаря | литерал [!]  
 | имя-шаблона [!]  
 | (элемент-шаблона {, элемент-шаблона}) [!]

Таким образом, каждый шаблон словарной единицы задается именем слева от знака «=» и собственно описанием, которое является последовательностью примыкающих (обозначение — знак «+») друг к другу фрагментов. Каждый из них может быть пустым, альтернативой NIL, ссылкой на словарь, на другой шаблон, литералом или списком альтернативных фрагментов. Последние три варианта могут повторяться (знак «!»).

Все фрагменты описания шаблона являются либо литералами, либо именами словарей. Собственно описание словарей содержится во второй секции раздела словарей — *секции значений словарей*. Здесь дается определение каждого словаря и рекомендация по отображению его на разные типы памяти, а также описываются отдельные словарные статьи. Для сокращения объема словарей словарные статьи могут объединяться в словарные группы. Ниже возможности словарной компоненты языка ATNL-2.0 иллюстрируются на примере описания фрагментов из секций шаблонов и значений словарей:

VOCAB-DIVISION.

VOCDS-SECTION.

СЛОВО = ОСНОВА + ОКОНЧ + (ПОСТФ, NIL);

ОСНОВА = (ПРЕФ, NIL) + КОРЕНЬ + (СУФ1, NIL);

ПОСТФ = («СЯ», «СЬ»);

VOCVL-SECTION.

VOC ПРЕФ IS-INTERNAL.

словарная-статья-1

словарная-статья-2.

словарная-статья-п.

VOC КОРЕНЬ IS-EXTERNAL; DB-IS ROOT.

структура-словарной-статьи

VOC ОКОНЧ.

словарная-группа-1  
 . . . . .  
 словарная-группа-2  
 MUST-EXTERNAL словарная-группа-т.  
 словарная-группа-(т+1).  
 . . . . .

*Словарная статья* содержит ключ, например конкретный префикс, корень или окончание, и список поименованных характеристик. Значение характеристики может быть литералом или последовательностью литералов, ссылкой на соответствующие характеристики другой словарной статьи или, наконец, функцией, значение которой является значением характеристики. Структура *словарной группы* аналогична, только общие характеристики не дублируются. Вместе с тем и в том, и в другом случае макроструктура словарной статьи или группы задается явным указанием номеров уровней, на которых расположены ключи и/или характеристики. Например, словарная группа из словаря ОКОНЧ может иметь вид

01 ТИП=«СУЩ»; ОДУШ=«НЕТ»; РОД=«М».  
 02 ЧИСЛО=«ЕД».  
   03 NIL ПАДЕЖ=«И», «В»; ...  
   03 «А» ПАДЕЖ=«Р»; ...  
 . . . . .  
 02 ЧИСЛО=«МН».  
   03 «И» ПАДЕЖ=«И»; ...  
 . . . . .

Если словарь задается структурой словарной статьи (как для внешнего словаря *КОРЕНЬ* из предыдущего примера), в рамках ее описания определяются названия характеристик, а также ограничения на их значения, которые хранятся в базе данных:

01 ТИП=«СУЩ», «ГЛАГОЛ», ...  
 02 ЧИСЛО=ANY.  
 03 РОД=«М», «Ж», «С».  
 04 КЕУ ПАДЕЖ; СЕМАНТ-ЭКЕРТ NIL; ...

Раздел нестандартных функций содержит две секции: *внутреннего и внешнего развития языка*. В первой вводятся обозначения для фрагментов ATNL-программы, повторяющихся в разных местах. Вторая секция позволяет добавить в программу Л-процессора блоки, написанные на других языках программирования. Раздел нестандартных функций факультативный в ATNL-программе.

В разделе сети описывается модель общения, а также алгоритмы анализа входных и синтеза выходных фраз, обрабатываемых Л-процессором. По существу, это ядро ATN-формализма. Различные версии ATN могут значительно отличаться друг от друга, однако все они имеют общий механизм движения по сети, общую идеологию использования памяти Л-процессора, а также базисную совокупность системных регистров, практически общую для всех реализаций. Как и в классическом ATN [Woods, 1970], в языке ATNL-2.0 описание сети задается совокупностью кустов (состояние и исходящие из него дуги). При этом каждая дуга описывает некоторую альтернативу, а все дуги вместе — функцию обработки информации в заданном состоянии. В языке ATNL-2.0 используется четыре типа дуг: TST, CAT, PUSH и POP. Структурно все они подобны друг другу и включают: прагматику дуги в виде ее наименования; условия применимости; действия по переработке информации; ссылку на состоя-

ние-приемник, быть может, неявную. Первые две компоненты составляют зону применимости дуги, а остальные — зону преобразования:

описание куста

имя-состояния дуга {дуга}

дуга

зона-применимости- $\gg$  зона-преобразования

зона-применимости

TST	условия-применимости
CAT	
PUSH	
POP	

зона-преобразования

{действие;}	заключительное-действие
форма	

Как следует из приведенных форматов, в языке ATNL-2.0 формализм расширенных сетей переходов синтаксически закреплен продукционным описанием. Это позволяет существенно повысить наглядность ATNL-программ по сравнению с классической Лисп-нотацией.

В общем случае в ATNL-2.0, как и в классическом формализме Вудса, условие применимости распадается на С- и Р-условия. Семантика С-условий различна для CAT- и PUSH-дуг. Для CAT-дуги это условие истинно, если текущая лексема текстуально совпадает с пометкой дуги или ее можно отождествить (по определенным правилам) с одним из слов в заданном словаре. Если поиск по словарю завершился удачно, характеристики лексемы становятся доступными для дальнейшего использования, для этого они переносятся в системный регистр WORD(\*). Ввиду наличия в естественных языках омонимии в ATNL-2.0 введен специальный регистр IMPR(&), где хранятся морфологические омонимы обработанных лексем. Таким образом, для CAT-дуги С-условие является предикатом с побочным эффектом обращения к словарю. Для PUSH-дуги это предикат, принимающий значение истинности тогда и только тогда, когда переход к указанному в условии подграфу успешно завершается выходом из него через соответствующую POP-дугу, т. е. для PUSH-дуги С-условие структурное. Заметим, что и для TST-, и для POP-дуги С-условие считается выполненным по умолчанию.

Основу зоны преобразования составляют в ATNL-2.0 действия и формы. Действия обеспечивают запоминание значений, полученных с помощью форм:

действие

имя-регистра := форма	
имя-регистра.DOWN := форма	
имя-регистра.UP := форма	
DOWN. имя-регистра	
UP. имя-регистра	
TRANSIT	

Так как в ATN-формализме все регистры — «слоистые» стеки, первая альтернатива SETR специфицирует присваивание значения на текущем уровне рекурсии, последующие две альтернативы служат для передачи значения на один уровень ниже (суффикс .DOWN) и выше (суффикс .UP) текущего. Эти действия эквивалентны SENDR и LIFTR соответственно. Последние два действия локализованы по эффекту выполнения внутри текущего уровня и синтаксически оформлены в виде префиксов DOWN. и UP. к имени регистра, на который действуют. Такие действия предназначены для «проталкивания» и

«вытalkingивания» стека с именем соответствующего регистра. Последним действием в языке ATNL-2.0 является действие TRANSMIT. Его аргументом служит некоторая форма, значение которой передается на вход подчиненного каскада ATN. Таким образом, в рамках ATNL-2.0 реализуются каскадные ATN [Woods, 1980].

Все формы в языке ATNL-2.0 делятся на две группы, обеспечивающие чтение и формирование значений. Чтение значений обеспечивается формами: GETS для словарных характеристик, GETR для значений регистров без сохранения, GETF для значений регистров с сохранением их в том же регистре, а также формами \*, @, #, &, которые идентифицируют соответствующие системные регистры. Для формирования значений в ATNL-2.0 введены формы явного задания значений, получения значения по его прототипу и задания композиций и структур из значений. В целом формы этой группы соответствуют формам классических реализаций ATN, а все отклонения либо носят синтаксический характер, либо связаны с желанием обеспечить разработчика Л-процессора простыми удобными средствами формирования требуемых структур.

Заключительные действия представлены в ATNL-2.0 тремя типами: JUMP — переход без считывания информации из входного потока; TO и OT — переход со считыванием информации слева направо или справа налево в верхушку системного регистра \*; GO — косвенный переход по значению # — регистра. Имя состояния, в которое происходит переход, может быть задано явно либо косвенно — путем указания имени хранящего его регистра или характеристики.

### Макропроцессор и язык MATNL

Язык ATNL-2.0 является дальнейшим развитием языка ATNL и ATNL/2 [Хорошевский, 1979] в сторону практически используемой системы представления лингвистических знаний. Применение языка ATNL на практике показало, что его базовые средства ориентированы в первую очередь на программирование ЕЯ-анализаторов и генераторов. Однако в настоящее время создание практически полезных модулей общения связывается с реализацией не только лингвистической модели языка общения, но и общей схемы диалогового взаимодействия человека и ЭВМ. А это предполагает использование нетривиальных, хотя и регулярных последовательностей ATNL-конструкций, описывающих различные схемы диалога. Учитывая вышесказанное, а также необходимость обеспечения проблемных специалистов, разрабатывающих модули общения, простыми, удобными и компактными средствами описания моделей диалога, представляется полезным введение в программное обеспечение разработки средств общения специализированных макросредств, отвечающих сформулированным требованиям. Поэтому в ATNL-систему и был введен описываемый ниже макропроцессор MATN, который, являясь специализированным, активно использует ориентацию на базовый язык ATNL-2.0.

Как и любой практически полезный микропроцессор, MATNL должен обеспечивать построение результирующего текста на базовом языке с использованием макроопределений из MATNL-программы и макроблиблотеки. Поэтому в макроязыке MATNL должны быть средства идентификации макроблиблотеки и определения макросов. Наиболее естественно ввести требуемые средства в разделы модульности и нестандартных функций. Тогда в предложениях идентификации внешних объектов в секции связи необходимо ввести элементарные типы MACRO и MCLIB, а описания макросов добавит в секцию внешнего развития языка ATNL-2.0. При этом получают такие определения:

описание-внешних-объектов

группа-типа {; группа-типа}

группа-типа

имя-объекта {, имя-объекта} WITH-TYPE тип-объекта.

|| IS-EXTRN ||  
|| ARE-EXTRN ||

где имя-объекта и тип-объекта, кроме введенных ранее, могут быть именем макроса или макроблиотеки и соответственно MACRO или MCLIB;

секция-внешнего-развития-языка

EXDEV-DIVISION || описание-макросов  
|| описание-интерфейса-с-другими-языками ||

описание-макросов

макрос {макрос}

макрос

MACRO прототип MCDEF макрорасширение MEND

Разработка простых и удобных, но вместе с тем мощных микросредств для проблемных специалистов является основной задачей. Конкретизация этой задачи выдвигает два основных требования к макроязыку и макропроцессору: определения макросов должны быть простыми и позволять пользователю гибко описывать требуемые расширения MATNL; макропроцессор должен обеспечивать хорошую диагностику ошибок на уровне, понятном пользователю. В MATNL используется идея синтаксических макросов Ливенворса [Leavenworth, 1966]. Для повышения мощности макроязыка введены макросы любого уровня вложенности и достаточно развитые средства периода генерации.

Макрос является основной декларативной конструкцией MATNL, а макровывод — процедурной. Макровыводы могут использоваться не только в разделах словарей и сети, но и в макроопределениях. Синтаксис описания прототипов очень прост и базируется на идее использования ограничителей, определяемых самим пользователем:

прототип  
[параметр-метка] имя-макро [{параметр}]  
параметр-метка  
&имя  
имя-макро  
@ имя  
параметр  
ограничитель {ограничитель} значение-параметра  
значение-параметра  
|| & имя ||  
|| "текст" ||

Такое решение требует, однако, введения специальных маркеров, идентифицирующих имена макро и значения параметров. Имя макро может быть произвольной последовательностью букв и/или цифр, которая начинается специальным символом @. Параметр может быть символом переменной (в этом случае ему предшествует &) или литералом (тогда он берется в кавычки). Ограничителем может быть любая последовательность символов доступного при реализации макропроцессора алфавита.

При описании прототипов, кроме рассмотренных средств, используются собственные знаки языка — круглые, квадратные и угловые скобки, а также символ </>. Если пользователю эти знаки потребуются использовать в другом смысле, каждому из них должна предшествовать двойная кавычка. Семантика собственных знаков языка следующая: конструкция, заключенная в круглые скобки, может повторяться любое число раз; конструкция, заключенная в квадратные скобки, необязательная; из элементов конструкции, заключенной в угловые скобки, один должен присутствовать обязательно (разделителем элементов служит </>). Таким образом, собственные знаки MATNL позволяют описывать прототипы с переменным числом параметров, с необязательными параметрами, а также вводить альтернативные ограничители. Все это повышает гибкость макроязыка и расширяет базовый язык.

Мощность макроязыка определяется не столько базовым языком, сколько доступными пользователю средствами периода генерации. В MATNL они пред-

ставлены символами периода генерации, а также системными макросами определения значений периода генерации, условной генерации и вывода. Для описания системных макросов воспользуемся рассмотренными выше средствами описания прототипов. Определение значений в MATNL осуществляется системным макросом @LET с прототипом вида

@LET &ПЕРЕМ=&ЗНАЧ (, &ПЕРЕМ=&ЗНАЧ);

где &ПЕРЕМ — имя локальной внутри текущего макрорасширения переменной, а &ЗНАЧ может быть целым числом или текстом. Для получения первого типа значений используются целые константы и знаки операций сложения «+» и/или умножения «\*». Текстовое значение получается с помощью литеральных констант и/или операции конкатенации «.». LET-макросы могут использоваться только в макрорасширениях.

Символы периода генерации предназначены в MATNL для ссылок на нужный элемент прототипа и определения меток периода генерации, используемых в соответствующих операторах. Указатель элемента прототипа маркируется &&P& ЗНАЧ, а метка периода генерации — &&L& ЗНАЧ. Операторы условной генерации представлены в MATNL макросами безусловного и условного переходов, а также макросом прерывания обработки макрорасширения. Их прототипы следующие:

@GOTO &МЕТКА;

@IF &УСЛ THEN &МЕТКА [ELSE &МЕТКА];

@EXIT

Параметр &МЕТКА должен быть меткой периода генерации, а параметр &УСЛ — композицией переменных периода генерации и их значений, соединенных знаками отношений «<», «<=», «>», «>=».

Макрос вывода в MATNL представлен единственным оператором периода генерации с прототипом вида

@NOTE &ТЕКСТ (, &ТЕКСТ);

параметр &ТЕКСТ может быть переменной периода генерации или литералом.

Все макросы периода генерации выполняются во время макрорасширения и в случае использования в качестве параметров переменных периода генерации локализованы в пределах текущего макрорасширения.

## **2.5. Средства для работы со знаниями в технологическом комплексе ТХК-БЗ**

*Е. Ю. Кандрашина*

В данном разделе приводится краткое описание двух языков продукционного типа, разработанных в ВЦ СО АН СССР в рамках проекта по созданию интегрального технологического комплекса баз знаний (ТХК-БЗ). Первый из них ЯСП реализован и эксплуатируется, второй (Л-язык) характеризует одно из направлений развития комплекса ТХК-БЗ.

### **Продукционный ЯСП**

Продукционный ЯСП реализован в составе технологического комплекса ТК-СП [Нариньяни и др., 1986], используется для спецификации комплекса систем продукции (СП) (см. § 3.1).

Особенностями ЯСП являются возможность формирования комплекса СП и наличие средств для организации двухуровневого динамического управления исполнением производственных правил в отдельной СП и комплексе. При этом роль базовой памяти выполняет семантическая сеть.

Необходимость в средствах спецификации комплекса СП возникает в случае, когда система представления и обработки знаний включает несколько СП, выпол-

няющих различные функции, в частности соответствующих различным этапам обработки информации или фрагментам проблемного знания.

Комплекс СП вводится как последовательность отдельных СП. Исполнение комплекса СП организуется с помощью двух специальных структур памяти: стека и очереди вызовов СП, в которых хранятся имена СП, подлежащих исполнению. Сразу после вызова комплекса СП составляющие его СП (точнее, их имена) в порядке перечисления заносятся в очередь вызовов, а стек вызовов остается пустым. Первыми для исполнения выбираются СП, находящиеся в стеке. После исчерпания стека исполняются СП, стоящие в очереди. Когда стек и очередь становятся пустыми, работа комплекса СП завершается.

Для организации работы комплекса в язык включены следующие операторы: ВЫЗОВ — вызов СП, ВЫЗОВСВ — вызов СП с возвратом, АКТСП — задержанный вызов СП. Все эти операторы могут включаться в правую часть продуктов.

Оператор ВЫЗОВ (СП<sub>i</sub>) прерывает исполнение текущей СП, после чего переходит к исполнению СП<sub>i</sub>. Оператор ВЫЗОВСВ (СП<sub>i</sub>) реализует вызов СП<sub>i</sub> с возвратом, при этом имя текущей СП и ее состояние в момент прерывания заносятся в стек, одновременно происходит активизация (вызов на исполнение) СП<sub>i</sub>. Оператор АКТСП(СП<sub>i</sub>) служит для «задержанного вызова» СП<sub>i</sub>: имя СП<sub>i</sub> помещается в очередь, при этом исполнение текущей СП не прерывается.

Система продукций в ЯСП вводится как именованное множество продукционных правил, допускающее структуризацию — выделение подмножеств. Из них одно объявляется начальным множеством активации, именно с него начинается исполнение СП. Таким образом, описание СП включает следующие разделы:

имя,  
наполнение — множество именованных продуктов,  
структура — совокупность именованных подмножеств  
продукций,  
имя начального множества активации.

Подмножества, образующие структуру СП, вводятся конструкцией вида  
 $M_i$  группа ( $p_1, \dots, p_k$ ),

где  $M_i$  — имя подмножества;  $p_1, \dots, p_k$  — имена продуктов, включенных в  $M_i$ .  
Информация о ходе исполнения СП содержится в переменных АЛОК и АГЛОБ.

Исполнение СП состоит в задании на каждом шаге ее работы множества активированных продуктов  $M_k$  и его исполнения. В начальный момент  $M_k$  совпадает с начальным множеством активации данной СП. Модификация  $M_k$  осуществляется посредством специальных операторов, включенных в состав продукций. Всякое изменение  $M_k$  немедленно переводит СП в следующий шаг ее исполнения, при этом переменным АЛОК и АГЛОБ присваивается значение 0 (это же значение присваивается переменным в самом начале работы СП).

Рассмотрим некоторый фиксированный шаг исполнения СП с множеством активации  $M_k$ . Он состоит из двух этапов:

1) исполнение продуктов из  $M_k$  в произвольном порядке до тех пор, пока хотя бы одна из них может быть исполнена, т. е. хотя бы у одной из продуктов выполнено условие применимости, при этом успешное применение любой продукции устанавливает признак АЛОК=1;

2) значение АГЛОБ меняется на 1 и осуществляется повторное исполнение  $M_k$ .

Если на текущем шаге не произошло модификации  $M_k$ , то исполнение СП считается завершенным.

Определяемые в ЯСП продукты исполняются над базовой памятью, представляющей собой конъюнктивную семантическую сеть со средствами структуризации. Данная сеть вводится как множество записей, которые отображают бинарные и/или  $n$ -арные отношения. Записи имеют следующую структуру:

$M: P(\text{arg}_1, \dots, \text{arg}_N)$

где М — метка вводимого отношения; Р — его имя;  $arg1, \dots, argN$  — аргументы. Метка является равноправной вершиной сети и обеспечивает возможность ее структуризации. В качестве аргументов отношения могут выступать объекты (синтаксически они соответствуют идентификаторам в языках программирования), целые числа, метки (помечаются звездочкой) и имена отношений (заклю- чаются в кавычки). При описании п-арных отношений наряду с аргументами указываются имена позиций, которые они занимают. Приведем примеры запи- сей:

«возраст» (Крылов, 35)  
 \*M2: «тип» (a1, человек)  
 \*I0: «элемент-подсети» (С, \*M2)  
 \*K1: «купить» ((кто, чел1), (что, книга), (у-кого, Вася))  
 «подтип» («приобрести», «купить»).

Продукционное правило в языке ЯСП имеет следующую структуру:  
 $M: P0 // P+(x1, \dots, xN); P-(y1, \dots, yN) //$

$$P(xy1, \dots, xyK) \Rightarrow Q(x1, \dots, xN),$$

где М — имя продукции;  $P0$  — независимое условие;  $P+$  — положительный об- разец;  $P-$  — отрицательный образец;  $P$  — зависимое условие;  $Q$  — последователь- ность действий (операторов), выполняемых продукцией. Компоненты  $P0$ ,  $P+$ ,  $P-$ ,  $P$  в комплексе образуют условие выполнимости продукции. Любая из час- тей условия может отсутствовать.

Независимое условие задается предикатом, определенным на переменных АЛОК и АГЛОБ, например  $(АЛОК=1) \& (АГЛОБ=1)$ .

Положительный образец  $P+$  соответствует фрагменту семантической сети, в котором отдельные вершины помечены как переменные (переменные метятся знаком \$), например:

$P+ =$  («тип» (\$ x1, человек), «тип» (\$ x2, человек),  
 «подтип» (\$ x3, «родственники»), \$ x3 (& x1, \$ x2)).

Отрицательный образец  $P-$  представляет собой набор таких же фрагментов. При этом множества переменных  $P+$  и  $P-$  могут пересекаться.

Зависимое условие  $P$  определено на переменных из положительного образца и представляет собой обращение к процедуре, вырабатывающей логическое значе- ние либо составной предикат, формируемый из некоторого набора элементар- ных предикатов (МЕНЬШЕ, РАВНО, БОЛЬШЕ и т. д.). Условие продукции считается выполненным, если  $P0$  истинно и в семантической сети есть хотя бы один фрагмент, удовлетворяющий образцу  $P+(x1, \dots, xN)$ , такой, что после подстановки найденных значений переменных  $x1, \dots, xN$  в остальные компонен- ты левой части продукции ( $P-$  и  $P$ ) в сети не удается найти ни одного фраг- мента из  $P-$  и  $P$  принимает значение «истина».

Исполнение продукции осуществляется последовательно для всех фрагментов семантической сети, для которых выполнено условие их применимости, и заклю- чается в реализации операторов из правой части продукции  $Q(x1, \dots, xN)$ , где предварительно производится замена переменных на соответствующие им вер- шинны из очередного фрагмента сети. В состав левой части могут входить опе- раторы следующих типов:

операторы языка СЕТЛ;  
 функции редактирования семантической сети (добавить, удалить фрагмент, склеить, удалить вершину);  
 функции ввода-вывода (печатать аргументы на экране, ввести строку с экра- на, записать объект в архив, считать объект из архива, ввести целое с экрана);  
 функции локальной активации — обеспечивают возможность динамического управления множеством активации (для обозначения этого множества использу- ется зарезервированное имя МТ) в процессе исполнения СП: присваивание, объ- единение, пересечение, разность множеств, включение продукции в множество, исключение продукции из множества;



функции глобальной активации ВЫЗОВ, ВЫЗОВСВ, АКТСП применяются для управления комплексом СП.

### Л-язык

Более современное, чем ЯСП, средство представления знаний — Л-язык. Он базируется на понятии *леммы*, которая задает своеобразную схему описания понятий предметной области и тем самым поддерживает определенную дисциплину, методологию формирования проблемного наполнения интеллектуальной системы.

В основу языка положена двухуровневая схема: собственно язык спецификации знаний — Л-язык для пользователя и базовая формальная система (БФС) для разработчика ядра интеллектуальной системы. Базовая формальная система — самостоятельный язык, который объединяет все три парадигмы представления знаний: логическую, структурную и процедурную. Семантика Л-языка полностью описывается в терминах БФС.

*Базовая формальная система* включает библиотеку понятий, функционально-семантическую сеть (ФС-сеть) и продукционную систему над ФС-сетью. Библиотека содержит описания классов (сортов) объектов, отношений и функций. Описания понятий включают два уровня: декларативный и интерпретационный (последний может отсутствовать). На первом вводится обозначение понятия, на втором — интерпретация: для класса объектов — носитель (множество значений), возможно, неполный; для отношений и функций — способ (процедура) их «вычисления». Описание функции дополнительно несет информацию о существовании и единственности ее значения при любом фиксированном наборе аргументов. Эта информация формирует структурный уровень ФС-сети. Элементами носителя могут быть числа, строки и структуры, построенные на их основе.

ФС-сеть объединяет возможности функциональной и семантической сетей. В ней выделяется три типа вершин: объекты, функции и отношения. Дуги отражают связи функций и отношений со своими аргументами. Каждая объектная вершина имеет два уровня представления: декларативный и интерпретационный. На первом с ней связывается класс (возможно, несколько) и имя, на втором — значение данного объекта или множество возможных значений. Предполагается, что БФС изначально содержит определение класса МНОЖЕСТВО, семантика которого считается встроенной. Тем самым объектная вершина может представлять как единичный объект, так и множество объектов. Функциональные вершины, а также отношения, имеющие интерпретацию, в рамках ФС-сети обладают собственной активностью. Отношения и функции, имеющие интерпретацию, вычисляют или корректируют значения своих аргументов подобно тому, как в общепринятых вычислительных моделях [Нариньяни, 1986]. Независимо от наличия интерпретации функции стремятся «склеить» объектные вершины, соответствующие их результатам, при условии тождественности аргументов. Такие локальные процессы коррекции активизируются при всяком изменении объектных вершин (в частности, при появлении новых) и исполняются вплоть до полной стабилизации ФС-сети. Допускается также структуризация ФС-сети — выделение фрагментов (подсетей), которые рассматриваются как самостоятельные вершины класса ПОДСЕТЬ. Различают открытые и закрытые подсети. В первом случае наполнение подсети считается включенным в объемлющую подсеть и доступно наравне с остальными ее элементами, во втором — содержащаяся в подсети информация считается изолированной и «добраться» до нее можно только через обращение к соответствующей подсети. Открытые сети, например, могут использоваться для отображения сообщений типа «в газете сообщалось, что ...» или для проблемной структуризации ФС-сети. Закрытые сети удобны при введении альтернативных фактов или отображении сообщений, достоверность которых вызывает сомнения, например таких, как «Мэри считает, что ...», «ей приснилось, что ...».

*Система продукций* в БФС вводится как некоторое расширение ЯСП. Наряду со средствами динамического управления исполнением продукций, заимствованными из ЯСП, в БФС существует возможность активации продукций из ФС-сети. Это обеспечивается следующим образом. Со всякой объектной верши-

ной ФС-сети, в частности подсетью, может быть связана продукция (множество продукции), которая активируется при изменении данной вершины. Таким образом можно осуществлять, например, локализацию исполнения группы продукции (их привязку к конкретному фрагменту ФС-сети).

Из-за возможности активации продукции из ФС-сети существуют некоторые отличия и в исполнении отдельной продукции. Покажем это на примере продукционного правила, отражающего всем известное свойство отношения порядка между точками на прямой  $x_1, x_2, x_3, x_4$ :

$$\begin{aligned} (x_1, x_2, x_3, x_4 \in \text{ТОЧКА}) \quad (x_1 < x_2) \quad (x_2 < x_4) \quad (x_1 < x_3) \\ (x_3 < x_4) \quad (y_1 = R(x_1, x_2)) \quad (y_2 = R(x_3, x_4)) \quad (y_3 = R(x_1, x_4)) \\ (y_4 = y_1 + y_2) \quad (y_3 < y_4) \Rightarrow x_3 < x_2, \end{aligned}$$

где  $y_1, y_2, y_3, y_4$  — переменные типа ЧИСЛО;  $R$  — функция расстояния.

Пусть с классом ЧИСЛО, функцией «+» и отношением «<», определенными над числами, соотносена интерпретация, а ТОЧКА и функция расстояния  $R$  интерпретации не имеют. При этом левую часть продукции можно разбить на две части — декларативную  $P_d$  и интерпретационную  $P_{int}$ :

$$\begin{aligned} P_d &= (x_1, x_2, x_3, x_4 \in \text{ТОЧКА}) \quad (x_1 < x_2) \quad (x_2 < x_3) \quad (x_1 < x_3) \quad (x_3 < x_4) \\ (y_1 &= R(x_1, x_2)) \quad (y_2 = R(x_3, x_4)) \quad (y_3 = R(x_1, x_4)); \\ P_{int} &= (y_4 = y_1 + y_2) \quad (y_3 < y_4). \end{aligned}$$

Проверка условий выполнимости данной продукции включает два этапа (результаты поиска объединяются): обычный поиск по образцу для всей левой части продукции и частичный поиск, состоящий в поиске образца  $P_d$  и вычислении  $P_{int}$ .

Л-язык как бы надстраивается над БФС за счет ввода терминальных выражений и специальных конструкций, обеспечивающих более компактное и удобное представление знаний. Специфической единицей Л-языка является лемама. С первого взгляда она очень напоминает фрейм, однако сходство чисто внешнее. При «трансляции» выражений Л-языка в БФС она трансформируется во множество определений библиотечки понятий и группу функций. Выделяется три типа лемм: для описания функций, отношений и классов объектов, незначительно отличающихся друг от друга составом и семантикой своих полей. Наиболее богата по содержанию объектная лемма (или о-лемма), включающая следующие компоненты (поля): имя; иерархический контекст; определитель; отрицательный определитель; интерпретацию; оболочку; функциональную схему оболочки и модель. Безусловно, в описание конкретного класса объектов не обязательно включается полный набор полей, любое из них может отсутствовать (конечно, с учетом взаимообусловленности их компонент).

Имя вводит обозначение описываемого класса и переменную, используемую в остальных полях как произвольный элемент данного класса, например имя: ЧИСЛО  $x$ ; имя: КНИГА  $u$ .

Иерархический контекст содержит перечень понятий, являющихся более общими и/или более частыми по отношению к данному. Так, для класса СОБАКА

нер-контекст:

родители ДОМАШНЕЕ-ЖИВОТНОЕ, ПЛОТояДНОЕ;  
дети ДВОРНЯГА, СЛУЖЕБНАЯ, ОХОТНИЧЬЯ, ДЕКОРАТИВНАЯ

Существует возможность помечать полиые альтернативно-исключающие разбиения класса.

Определитель включает достаточные условия принадлежности элементов классу и/или его точное определение. В рамках данного поля можно использовать определения двух типов:

предикатно-сужающее — накладывает ограничения на класс или множество классов из поля «родители», например лемма, описывающая класс РЕБЕНОК, может содержать определение

п-опред: ЧЕЛОВЕК  $x$ ; возраст( $x$ ) < 12 лет;

интерпретируемое следующим образом:

$(x \in \text{ЧЕЛОВЕК}) \& (\text{возраст}(x) < 12 \text{ лет}) \Rightarrow x \text{ РЕБЕНОК};$

предикатно-коистроующее — описывает способ построения объектов типа Ао (вводимых данной леммой), например:

к-опред:  $(x_1, x_2, x_3 \in \text{ТОЧКА}) \& \text{П1}(x_1, x_2, x_3) \Rightarrow$   
 $\Rightarrow (\exists x) (x \hat{=} \text{ТРЕУГ}(x_1, x_2, x_3)).$

где П1 — предикат «не лежат на одной прямой»;  $x$  — переменная из поля имя данной леммы, описывающей класс ТРЕУГ — треугольник.

Отрицательный определитель содержит условия непринадлежности объектов данному классу, например для класса СОБАКА:

и-опред:  $(x \in \text{ДОМАШНЕЕ-ЖИВОТНОЕ}) \& \text{мяукает}(x).$

Такая конструкция, в частности, может использоваться для вывода от противного.

*Интерпретация* вводит множество значений (иоситель) описываемого сорта.

Оболочка, являющаяся аналогом системы слотов во фрейме, представляет собой совокупность функций и отношений некоторого специального вида: класс одного из аргументов должен совпадать с описываемым классом Аі. Оболочка выделяет своего рода семантическую окрестность (когнитивное пространство) объекта класса Аі и предоставляет синтаксические возможности для прямого доступа к своим элементам. Допускаются элементы оболочки (их по привычке будем называть слотами), порождаемые унарными функциями, бинарными и тернарными отношениями (предикатами). Тем самым выделяется три типа слотов: функциональные, предикатные и параметрические. Функциональный (наиболее типичный) слот соответствует унарной функции, определенной на описываемом классе, например год-рождения человека, стоимость вещи, название книги. В общем случае значением предикатного слота является корректируемое множество объектов. Его описание наряду с указанием имени и класса включает функцию его коррекции, например сторона треугольника (значение — множество из трех отрезков), последнее-место-работы (значение меняется при поступлении новой, более свежей информации). Параметрические слоты индуцируются тернарными отношениями. При этом один из аргументов должен принадлежать описываемому классу, второй выделяется как параметр и выносится в заголовок поля оболочки, третий выступает в качестве слота. Примерами таких слотов могут служить: возраст, дети, должность, вес, рост человека, где в роли параметра выступает время. Вообще говоря, с объектом может связываться несколько оболочек, соответствующих различным сферам (подмоделям) знаний, выделяемым в предметной области.

Функциональная схема оболочки задает условия однозначности определения объекта описываемого класса и/или значений его слотов.

Модель — совокупность отношений и продукций, связывающих элементы оболочки, отражает внутренние свойства описываемого понятия и при переходе к БФС трансформируется во фрагмент СП.

Заметим, что модель и функциональная схема соотносятся не столько с классом, сколько с приписываемой ему оболочкой.

Семантику двух последних полей продемонстрируем на следующем примере описания класса ИНТервал — одного из понятий формальной модели времени [Каидрашина, 1986].

имя: ИНТ t  
оболочка:  
начало; ТОЧКА x  
конец; ТОЧКА y  
длина; ЧИСЛО g  
ф-схема:  
 $x, y; x, g; y, g \longrightarrow t$   
модель:  $x < y$   
 $g = P(x, y)$

Данная ленима вводит класс объектов ИНТервал, а также три функции: начало, конец, длина, определяемые на этом классе. В функциональной схеме (ф-схеме) указано, что значения любой пары слотов однозначно определяют ИНТервал. Наполнение модели эквивалентно утверждению

$$t \in \text{ИНТ} \Rightarrow (\text{начало}(x) < \text{конец}(y)) \text{ (длина}(t) = \\ = R(\text{начало}(t), \text{конец}(t))).$$

Ленимы, задающие схемы описания функций и отношений, имеют практически тот же набор полей, лишь немного модифицируется их семантика.

Пока осуществлена единственная экспериментальная реализация некоторого подмножества Л-языка [Нариньяни и др., 1986], а также получен некоторый опыт его использования (при описании формальной модели времени [Каидрашина, 1986]). Безусловно, этого далеко недостаточно, чтобы объективно оценить достоинства и недостатки предлагаемых средств. Поэтому наиболее остро в настоящее время стоит проблема реализации полной версии Л-языка и накопления опыта его использования для спецификации моделей разнообразных сложных предметных областей.

## Глава 3.

# Инструментальные средства для разработки интеллектуальных систем

## 3.1. Технологические комплексы для разработки баз знаний

*Е. Ю. Кандрашина*

Технологические комплексы предназначены для разработки проблемно-ориентированных баз знаний различных типов; ТК-СП для построения семантических процессоров и ТК-ОВМ для построения процессоров обобщенных вычислительных моделей (ОВМ-процессоров). Первый базируется на концепции, которую можно специфицировать как «семантическая сеть плюс система продукций», предназначен для разработки базы знаний (БЗ) продукционно-сетевых типа. В основе второго комплекса лежит понятие ОВМ, обогащенное фреймами специального вида, его парадигма может быть определена как «функциональная сеть плюс фреймы». ТК-ОВМ поддерживает процесс разработки БЗ расчетно-логического типа. Вместе ТК-СП и ТК-ОВМ образуют первую версию интегрального технологического комплекса ТХК-БЗ. Комплексы имеют примерно одинаковую архитектуру. Каждый из них представляет собой параметризованную БЗ соответствующего типа, окруженную подсистемами спецификации ее структуры и проблемного наполнения. Комплексы предоставляют пользователю удобную среду для формирования, редактирования и отладки специфицируемых компонент БЗ. После завершения спецификации сформированная БЗ отторгается от комплекса и может быть использована как ядро или один из компонентов разрабатываемой интеллектуальной системы. Разработка такой системы с использованием комплекса включает следующие этапы:

- 1) выбор подходящей парадигмы представления знаний на основе анализа постановки задачи и характера предметной области;
- 2) разработка модели предметной области;
- 3) спецификация структуры процессора, создаваемого на основе комплекса, с учетом особенностей данной модели;
- 4) настройка формируемого процессора на предметную область (ввод и отладка модели предметной области);

5) отторжение готового процессора и включение его в состав разрабатываемой интеллектуальной системы (в простейшем случае подключение специализированной интерфейсной оболочки).

Комплексы ТК-СП и ТК-ОВП поддерживают этапы 3 и 4. Комплексы реализованы и эксплуатируются в лаборатории искусственного интеллекта ВЦ СО АН СССР. Их разработка проведена в рамках временного научно-технического коллектива СТАРТ. Текущие версии комплексов функционируют на ПЭВМ класса Labtam в операционной системе UNIX.

### Технологический комплекс ТК-СП

ТК-СП предназначен для создания проблемно-ориентированных производственных модулей, в которых роль базовой памяти выполняет семантическая сеть (СемС) [Загорулько и др., 1986; Загорулько, 1987; Нариньяни и др., 1986]. Модуль данного типа будем называть семантическими процессорами или С-процессорами. ТК-СП поддерживает технологию, при которой С-процессоры получают из некоторого типового С-процессора в результате спецификации его структуры и проблемного наполнения. Комплекс реализован на языке Паскаль с использованием автокода ГАММА-1 [Левин и др., 1986], отдельные компоненты — на языке Си. Общий объем его программной части около 20 тыс. строк.

К настоящему времени средствами комплекса разработано несколько демонстрационных интеллектуальных систем производственного типа. Первая из них МАРШРУТ является «экспертом» по правилам дорожного движения: в ходе подготовки системы к работе пользователь на готовой схеме автомобильных дорог (например, города) может по собственному усмотрению расставить дорожные знаки, после чего система сама проложит маршрут, оптимальный по качеству и продолжительности между любыми двумя точками схемы. Система ТЕЩА «знает все» о родственных отношениях. Функционально она соответствует логической базе данных, «разбирающейся» в системе родственных связей типа «родители», «золовка», «теща», «брат», «сестра» и т. п.

**Архитектура ТК-СП.** Общая схема ТК-СП изображена на рис. 3.1.

Типовой С-процессор включает компоненты двух типов: постоянные и специфицируемые. К постоянным относятся процессор свойств, интерпретатор команд интерфейсного модуля (ИКИМ), машина для оперирования СемС (С-машина) и процессор систем производций (СП-процессор). Специфицируемыми компонентами являются система подстановок и сценарий исполнения СП (СИСП) интерфейсного модуля, комплекс СП, комплекс свойств СемС. Интерфейсный модуль выполняет функцию связи С-процессора с конечным пользователем или внешними подсистемами: анализирует входной поток команд, поступающих в С-процессор, и отдает их на исполнение соответствующему модулю — СП-процессору, С-машине или процессору свойств. СП-процессор организует исполнение комплекса СП, С-машина предназначена для оперирования семантической сетью, являющейся основной памятью С-процессора, процессор свойств осуществляет коррекцию семантической сети в соответствии с ее свойствами.

Рассмотренная схема типового С-процессора соответствует максимальной конфигурации. Обязательными его компонентами являются лишь интерфейсный модуль (без специфицируемых компонент), С-машина и СемС. Они образуют С-процессор минимальной конфигурации — базовый. Базовый процессор в зависимости от потребностей пользователя может пополняться следующими наборами компонентов в произвольном сочетании: комплекс СП и СП-процессор, возможно, совместно с СИСП; система подстановок; комплекс свойств СемС и процессор свойств.

Модули спецификации предназначены для формирования проблемного наполнения конструируемого С-процессора. Его структура формируется неявно, в зависимости от наполнения специфицируемых компонент.

Перечисленные средства (наряду со средствами комплексирования и отладки) объединены в общую систему на уровне конструкторского монитора (К-монитора), обеспечивающего высокую технологичность процесса производства проблемно-ориентированных С-процессоров.

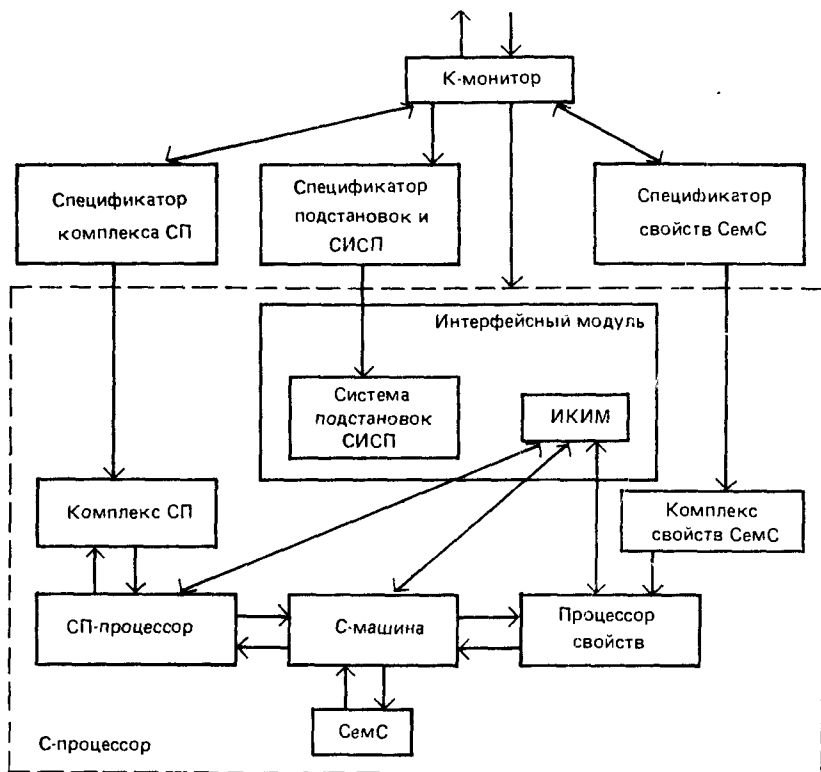


Рис. 3.1. Общая структура ТК-СП

**Базовые средства.** Базовые средства, образующие С-процессор минимальной конфигурации, представлены базовой семантической сетью (БСС), С-машиной и интерфейсным модулем. БСС представляет собой совокупность *элементарных компонент* (помеченных бинарных отношений) следующего вида:

$$m: r(\text{arg1}, \text{arg2}),$$

где  $m$  — метка отношения;  $r$  — имя отношения;  $\text{arg1}$ ,  $\text{arg2}$  — аргументы отношения. Особенностью БСС является то, что метка  $m$  однозначно именует соответствующий ей экземпляр бинарного отношения и может быть использована для ссылки на него. Элементарные компоненты несмотря на свою простоту обладают достаточной выразительностью и гибкостью; они позволяют, в частности, представлять в сети многоместные отношения, моделировать вложенность [Zagorkulko, 1984] и дизъюнктивность.

В базовой С-машине основным типом данных является БСС. Кроме БСС, она обрабатывает данные следующих типов: числа, строки, логические значения (ИСТИНА, ЛОЖЬ и НЕОПределенность), кортежи, множества, имена, элементарные компоненты, фрагменты, образцы. Числа, строки, логические значения, кортежи и множества являются данными виртуальной машины ГАММА-1. Имя — это последовательность символов, начинающаяся с буквы, используется для обозначения элементов БСС (при конструировании элементарных компонентов). Часть имени, выделяемая для использования в качестве переменных (будем на-

зывать их С-переменными), трактуется командами С-машины особым образом. Фрагмент представляет собой совокупность (множество) элементарных компонентов. Образец есть структура (кортеж), состоящая из двух элементов (второй может отсутствовать): фрагмента и кортежа фрагментов. Любой из фрагментов, входящих в образец, может включать С-переменные.

С-машина поддерживает исполнение трех групп команд: команды общего назначения, команды оперирования БСС, а также команды описания и редактирования данных. Команды первой группы служат для организации управления в программах (условные и безусловные переходы, вызов подпрограмм и т. д.), а также для работы с такими традиционными типами данных, как числа, строки, множества и кортежи. Для оперирования БСС С-машина предоставляет следующие команды: команду поиска по образцу, команды редактирования и команды для работы с архивом СемС, который входит в состав С-машины. По команде поиска по образцу выполняется поиск в сети фрагментов, удовлетворяющих указанному образцу. При этом первый элемент образца рассматривается как фрагмент поиска, именно он сопоставляется с сетью, а второй — как отрицательный контекст для первого (ни один фрагмент отрицательного контекста не должен быть сопоставлен ни с одним фрагментом БСС). Команды редактирования позволяют добавлять в БСС фрагменты, исключать из нее вершины и фрагменты (заданные явно или с помощью образца), склеивать и отождествлять вершины. Команды для работы с архивом позволяют сформировать в архиве новую сеть, уничтожить уже имеющуюся, настроить С-машину на работу с выбранной сетью (переписать ее в БСС). Команды описания и редактирования данных предназначены для описания и редактирования образцов и фрагментов, которые могут создаваться не только из элементарных компонентов, но и из многоместных ( $n$ -местных) отношений, определенных при спецификации свойств сети. При отображении в БСС каждый экземпляр  $n$ -местного отношения представляется  $n+1$  элементарный компонент.

Интерфейсный модуль реализован с учетом использования в составе любой конфигурации С-процессора. Он предоставляет пользователю достаточно удобный набор директив для оперирования семантической сетью (поиск информации, редактирование сети) в диалоговом или пакетном режиме, отображаемых в соответствующие команды С-машины, СП-процессора и процессора свойств. Одно из основных достоинств интерфейсного модуля — обеспечение возможности автоматического вызова и исполнения СП во время выполнения директив в соответствии с введенным сценарием.

**Средства спецификации.** Каждый из модулей спецификации имеет собственную архивную систему и предоставляет пользователю специализированный язык: ТИС — язык спецификации свойств семантической сети, ЯСП — язык спецификации СП, а также средства настройки интерфейсного модуля.

Для спецификации комплекса СП, включаемого в С-процессор для организации логического вывода [Нарньянн и др., 1987], служит ЯСП, предоставляя средства для спецификации продукционных правил и схемы их активации, позволяя организовать взаимодействие СП, входящих в комплекс (см. § 2.5).

С помощью языка ТИС строится проблемно-ориентированная семантическая сеть с заданными свойствами лишь на уровне внешних спецификаций, обеспечивающая более высокий уровень представления информации и контроль за ее правильностью.

Язык ТИС включает средства, с помощью которых конструктор, формирующий С-процессор, может выполнять следующие технологические операции:

1. Вводить классы отношений и объектов (допускается пересечение и вложение классов), а также определять, элементы каких классов могут выступать в качестве тех или иных аргументов отношений. Класс вершин может быть задан либо перечислением входящих в него вершин, либо конструированием новых классов из уже созданных с помощью традиционных теоретико-множественных операций: объединения, пересечения, дополнения. Кроме того, конструктор имеет возможность указать, что в сети могут использоваться только объекты и/или отношения, введенные в описания сети.

2. Предписывать отношению одно или несколько свойств (при условии, что они непротиворечивы) из следующего набора: СИМметричность, ТРАНСзитивность, РЕФлексивность, ЭКВивалентность, АНТИСИМметричность, АНТИРЕФлексивность, УНИКальность и ФУНКциональная Зависимость (ФУНКЗ). Семантика первых шести свойств понятна из их названий, поэтому поясним только семантику двух последних. Свойства УНИК и ФУНКЗ накладывают ограничения на второй аргумент отношения R. Так, при наличии в сети отношения R(A, B) добавление в сеть отношения R(A, C) вызовет АВОСТ, если R приписано свойство ФУНКЗ, и отождествление (склейку) вершин B и C, если R приписано свойство УНИК. Перечисленные выше свойства могут быть приписаны только бинарным отношениям или классам бинарных отношений, определенным с помощью тегиологической операции I.

3. Задавать интерпретации отношений в виде пакета продукционных правил. Прагматически интерпретация отношения представляет собой последовательность действий по модификации сети, выполняемую при добавлении в нее конкретного экземпляра такого отношения (при вводе информации через интерфейсный модуль или в процессе исполнения комплекса СП).

4. Определять начальное наполнение сети в виде некоторого стандартного фрагмента. Такой фрагмент может содержать, например, некоторые знания о предметной области и/или специальные знания, которые невозможно или нецелесообразно представлять с помощью других средств спецификации.

Настройка интерфейсного модуля предполагает спецификацию подстановок и СИСП. Средствами сценария с любым оператором пользовательского монитора может быть соотнесена СП, которая в рамках С-процессора будет исполняться при каждом обращении к данному оператору. Механизм подстановок аналогичен макросам в языках программирования. Благодаря ему пользователь при работе в СемС получает возможность манипулировать макроотношениями. Подстановка в общем виде выглядит следующим образом:

$$r_0(x_1, x_2) \rightarrow r_1(y_1, y_2), \dots, r_n(y_{2n-1}, y_{2n}),$$

где  $r_0, \dots, r_n$  — имена отношений;  $x_1$  и  $x_2$  — переменные;  $y_1, \dots, y_{2n}$  — вершины сети (имена) или переменные, причем  $x_1$  и  $x_2$  принадлежат  $(y_1, \dots, y_{2n})$ .

Перед компиляцией тексты программ (спецификаций) обрабатываются специальными препроцессором. Во время этой обработки каждый экземпляр отношения  $r_0$ , для которого задана подстановка, заменяется на группу отношений, указанных в правой части подстановки. При этом вместо переменных  $x_1$  и  $x_2$  подставляются соответствующие им аргументы обрабатываемого экземпляра отношения  $r_0$ . Механизм подстановок значительно повышает уровень спецификации данных в С-процессоре, а следовательно, и всего интерфейса.

При спецификации СИСП пользователь может связать с любой введенной ранее СП один из следующих признаков, определяющих «время» их исполнения:

ДОБ — добавление фрагмента;

ИСКЛ — удаление фрагмента;

СКЛ — склейка вершин;

МОД — любая модификация сети;

ПОИСК — перед поиском по образцу;

ДЕЖ — при любом обращении к сети;

НЕТ — СП не имеет признака и может быть вызвана только явно (по команде пользовательского монитора ВЫПОЛНИТЬ СП или с помощью функции ВЫЗОВ языка ЯСП).

**Пример создания С-процессора.** Технологический цикл создания проблемно-ориентированного С-процессора включает следующие этапы: описание свойств СемС, спецификацию интерфейсного модуля, создание комплекса систем продукций, сборку и тестирование С-процессора. Рассмотрим его на примере классической предметной области — родственные отношения (пример заимствован из работы [Загорулько и др., 1986]).



1. Описание свойств семантической сети. Введем следующие классы вершин:

ОТНОШЕНИЯ = («ПОЛ», «СТАРШЕ», «РОДИТЕЛЬ», «СУПРУГИ»);  
ДЕТИ = (Вася, Сережа, Маша, Катя);  
ВЗРОСЛЫЕ = (Иван, Петр, Татьяна, Ольга);  
ЛЮДИ = ДЕТИ + ВЗРОСЛЫЕ;  
СВОЙСТВА = (мужской, женский).

Опишем свойства отношений:

«РОДИТЕЛЬ» : структура  $\text{arg1} = \text{ВЗРОСЛЫЕ}$ ,  $\text{arg2} = \text{ДЕТИ}$ ;  
свойства {АНТИСИММЕТРИЧНОСТЬ};  
интерпретация

$\Rightarrow$  «СТАРШЕ» ( $\text{arg1}, \text{arg2}$ ).

«СУПРУГИ» : структура  $\text{arg1} = \text{ВЗРОСЛЫЕ}$ ,  $\text{arg2} = \text{ВЗРОСЛЫЕ}$ ;  
свойства {СИММЕТРИЧНОСТЬ}

«СТАРШЕ» : структура  $\text{arg1} = \text{ЛЮДИ}$ ,  $\text{arg2} = \text{ЛЮДИ}$ ;  
свойства {ТРАНЗИТИВНОСТЬ}

«ПОЛ» : структура  $\text{arg1} = \text{ЛЮДИ}$ ,  $\text{arg2} = \text{СВОЙСТВА}$ ;

Таким образом, свойства сети описаны, т. е. сконструирована проблемно-ориентированная СемС, включающая четыре базовых отношения, с помощью которых могут быть представлены следующие необходимые нам родственные отношения: «МУЖ», «ЖЕНА», «ОТЕЦ», «МАТЬ», «СЫН», «ДОЧЬ», «ДЯДЯ», «ТЕТЯ», «ПЛЕМЯННИК», «ПЛЕМЯННИЦА», «ЗЯТЬ», «НЕВЕСТКА», «ТЕЩА», «ТЕСТЬ», «ШУРИН», «ДЕВЕРЬ», «ЗОЛОВКА», «СВОЯЧЕНИЦА», «СВОЯК» и т. п. Такой способ представления, с одной стороны, позволяет более компактно хранить информацию, а с другой — существенно упрощает логический вывод. Однако для оперирования такой сетью требуется соответствующий интерфейс.

2. Спецификация интерфейсного модуля. Для повышения уровня ввода информации через пользовательский монитор определим следующие подстановки ( $x, y, z$  — переменные):

«МУЖ»( $x, y$ )  $\rightarrow$  «СУПРУГИ»( $x, y$ ), «ПОЛ»( $x$ , мужской);  
«СЫН»( $x, y$ )  $\rightarrow$  «РОДИТЕЛЬ»( $y, x$ ), «ПОЛ»( $x$ , мужской);  
«ВНУК»( $x, y$ )  $\rightarrow$  «РОДИТЕЛЬ»( $x, z$ ), «РОДИТЕЛЬ»( $z, x$ ),  
«ПОЛ»( $x$ , мужской);  
«ТЕЩА»( $x, y$ )  $\rightarrow$  «СУПРУГИ»( $y, z$ ), «ПОЛ»( $y$ , мужской);  
«РОДИТЕЛЬ»( $x, z$ ) «ПОЛ»( $x$ , женский);

Эти подстановки будут использоваться как при вводе информации, так и при обработке запроса.

3. Создание комплекса СП. В С-процессор включим комплекс СП, который будет состоять из двух СП. «Входная» СП активируется при вводе информации в сеть, а «выходная» — при ответе на запрос, требующий логического вывода. «Входная» СП содержит правила, которые дополняют и перестраивают сеть (поэтому они и активируются при вводе информации). Такие правила, например, определяют пол одного из супругов, отождествляют родителей (детей) по тем или иным признакам и т. п. Приведем два таких правила:

правило 1:  
//«СУПРУГИ»( $x, y$ ), «ПОЛ»( $x, s$ ); «ПОЛ»( $y, s1$ )//  
 $\Rightarrow$  если  $s$ -мужской то ДОБАВИТЬ («ПОЛ»( $y$ , женский))  
иначе ДОБАВИТЬ («ПОЛ»( $y$ , мужской))  
конец:  
правило 2:  
//«СУПРУГИ»( $x, y$ ), «РОДИТЕЛЬ»( $x, z$ ); «РОДИТЕЛЬ»( $y, z$ )//  
 $\Rightarrow$  ДОБАВИТЬ («РОДИТЕЛЬ»( $y, z$ ))  
конец.

где  $x, y, s, s1, z$  — С-переменные; компоненты «ПОЛ»( $y, s1$ ) и «РОДИТЕЛЬ»( $y, z$ ) соответствуют отрицательным образцам.

«Выходная» СП служит для формирования ответов на запросы вида «В каком родстве состоят Вася и Маша?», т. е. она активируется, когда требуется определить имя отношения, которое представлено в сети виртуально (например, «БРАТ», «СЕСТРА», «ШУРИН»). Эта СП содержит правила, представляющие собой фактически перевёрнутые подстановки, описанные выше.

4. Сборка и тестирование. Вначале С-процессор может собираться без комплекта СП. Такая конфигурация позволяет отлаживать описание свойств СемС и спецификацию подстановок. На следующем шаге к С-процессору подключаются и тестируются (сначала автономно, а потом в комплексе) СП, например сначала «входная» СП, за ней «выходная», а затем обе вместе. Только после выполнения этих манипуляций С-процессор может быть передан пользователю.

Если теперь на вход полученного С-процессора будет подана информация «МУЖ» (Петр, Ольга) и «СЫН» (Вася, Петр), то после выполнения соответствующих подстановок и работы «выходной» СП будут выведены и добавлены в сеть следующие отношения:

«СУПРУГИ» (Петр, Ольга);	«ПОЛ» (Петр, мужской),
«ПОЛ» (Ольга, женский),	«РОДИТЕЛЬ» (Петр, Вася),
«ПОЛ» (Вася, мужской),	«РОДИТЕЛЬ» (Ольга, Вася),
«СТАРШЕ» (Петр, Вася),	«СТАРШЕ» (Ольга, Вася).

В результате С-процессор сможет ответить на вопросы: «Кто отец (мать) Васи?», «Кто жена (сын) Петра?», «Кто старше — Петр или Вася?», а благодаря «выходной» СП и на вопросы типа «В каком родстве состоят Ольга и Вася?», «Кем доводится Ольга Петру?» и т. п.

### Технологический комплекс ТК-ОВМ

ТК-ОВМ предназначен для формирования проблемно-ориентированных ОВМ-процессоров, основу которых составляет потоковый вычислитель с общей памятью, реализующий ОВМ [Нариньяни и др., 1986]. Входной язык процессора определяется в ходе спецификации его проблемного наполнения, т. е. в рамках ТК-ОВМ.

Сформированный средствами комплекса ОВМ-процессор позволяет оперировать моделями объектов или явлений выбранной предметной области в достаточно удобной и естественной форме. Модель вводится как набор параметров (переменных того или иного типа) и связей (отношений) между ними. Параметрам ОВМ могут быть сопоставлены точные или недоопределенные значения [Нариньяни, 1986]. ОВМ-процессор автоматически поддерживает расчет ОВМ, в частности осуществляет коррекцию (уточнение) недоопределенных значений включенных в нее параметров, а также предоставляет удобные средства для ее редактирования. Это позволяет использовать ОВМ-процессор не только как модуль обработки знаний в интеллектуальных системах, ориентированных на решение конкретных задач в фиксированной предметной области, но и как средство для разработки собственно моделей, т. е. для автоматизации научных исследований. ТК-ОВМ реализован на языке Паскаль. Общий объем программ около 15 тыс. строк.

**Основные понятия.** В основу операционной части ОВМ-процессора положено понятие *обобщенной вычислительной модели функциональной сети со специальной недетерминированной схемой вычислений, обеспечивающей обработку обычных и недоопределенных значений переменных*. ОВМ оперирует базовыми типами данных и операторами, что не всегда позволяет компактно и наглядно описать модель исследуемого объекта. Поэтому с целью повышения уровня спецификации задач вводятся понятия *активного типа данных и отношения*. Первое позволяет определять достаточно сложные по структуре типы данных со встроенной операционной семантикой, второе играет роль, аналогичную макросам в языках программирования — идентифицирует параметризованные фрагменты ОВМ и фактически представляет собой ОВМ-процедуру.

На концептуальном уровне ОВМ определяется как двудольный ориентированный помеченный граф (ОВМ-сеть) в комплексе с некоторой недетерминированной дисциплиной его исполнения. В ОВМ-сети выделяется два типа вершин: объекты и операторы. С каждой объектной вершиной связывается тип (базовый), значение и функция присваивания, определяющая способ записи очередного значения в данную объектную вершину. С каждой операторной вершиной соотношены процедура, ее вычисляющая, целое число, играющее роль приоритета, и некоторая разметка входящих и исходящих из нее дуг. Выделяется два типа операторных вершин: операторы-функции и операторы-предикаты. Последние в ходе исполнения ОВМ осуществляют контроль за правильностью вычислений.

Дуги связывают операторные и объектные вершины. Входящие в оператор дуги соотносят с ним его аргументы, исходящие указывают на объекты, в которые должна производиться запись вырабатываемых им значений (допустимо наличие двунаправленных дуг). Символом \* отмечаются дуги, по которым осуществляется активация операторных вершин. Пометка  $\#$  на дуге  $l = (v_1, v_2)$ , где  $v_1, v_2$  — операторная и объектная вершины, означает, что запись значения в  $v_2$  должна производиться через соотнесенную с ней функцию присваивания. Наличие на одной дуге обеих пометок запрещено.

По сути, исполнение ОВМ-сети носит недетерминированный характер и, безусловно, в случае реализации данного процесса на последовательной ЭВМ можно говорить лишь о некоторой его имитации. Так как речь идет о конкретной реализации ОВМ, процесс ее исполнения опишем в том виде, в каком он реализован в рассматриваемой версии ТК-ОВМ. Для этого понадобится еще одно дополнительное понятие — очередь, которая используется для записи, хранения и вызова на исполнение операторов ОВМ-сети.

На каждом шаге исполнения ОВМ-сети из очереди выбирается и вычисляется первый из операторов с максимальным приоритетом. Полученные в результате вычисления значения записываются в соответствующие объектные вершины непосредственно или через функцию присваивания (при наличии пометки  $\#$  на соответствующей дуге). Если в результате записи какие-либо объекты изменили свои значения, то для каждого из таких объектов проверяются связанные с ним предикаты (при появлении хотя бы одного ложного значения объект исключается из дальнейших вычислений). Для объектов, изменивших свои значения и не исключенных из процесса вычисления, ищутся и помещаются в очередь операторы-функции, связанные с ними дугами с пометкой \*. Начальная инициация исполнения ОВМ-сети осуществляется посредством явного или неявного (указанием объектных вершин, изменивших свои значения) заполнения очереди.

Активный тип данных (АктД) можно рассматривать как абстрактный тип данных со встроенной операционной семантикой [Нарныни, 1986], используемый для спецификации сложных структурных объектов. Он включает множества слотов, условий на слотах, функций на слотах и допустимых операций над объектами описываемого типа. Каждому слоту в спецификации АктД сопоставлены имя слота, АктД, определенный ранее, или базовый тип, задающий тип значения данного слота, функция присваивания и начальное значение слота. Отсутствие функций присваивания или начального значения в описании слота подразумевает их наследование из упомянутого в слоте АктД или базового типа.

В качестве примера приведен АктД, описывающие понятия Н-ЧИСЛО (недоопределенное число) и КВАДРАТ. Н-ЧИСЛО — это приблизительно известное число, находящееся в пределах <нижняя граница, верхняя граница>. Нижняя граница Н-ЧИСЛА может только увеличиваться, а верхняя — только уменьшаться. Операции для Н-ЧИСЕЛ определяются в соответствии с правилами интервальной математики [INTMATH, 1975], соответствующий ему АктД определен так, что в процессе вычислений Н-ЧИСЛО может уточняться. Пусть  $m$  и  $M$  — наименьшее и наибольшее числа, допустимые в инструментальном языке, соответственно;  $\max$  и  $\min$  — функции, которые из двух значений выбирают наибольшее ( $\max$ ) и наименьшее ( $\min$ ). При этом АктД, соответствующий понятию Н-ЧИСЛО, будет иметь следующий вид (описание операций опущены, «вещ» — базовый тип, соответствующий вещественным числам):

Операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $**$ ,  $\text{sqrt}$  и др.

Определенное таким образом понятие Н.ЧИСЛО, в частности, позволяет организовывать обработку систем уравнений, а также строить более сложные недоопределенные типы данных. На его основе, например, можно определить АКТД КВАДРАТ с недоопределенными значениями параметров:

10. 1/2, 1/2, 1/2

Отношение определяется как параметризованный фрагмент ОВМ-сети. Его описание содержит:

множество функций (операторов) — описание функции включает ее приоритет, имя и список параметров — элементов из списка аргументов и/или значения их слотов (указывается порядковый номер аргумента и при необходимости через точку номера соответствующих слотов); параметром функции может быть только объект базового типа, при параметре может стоять одна из меток #, \*, интерпретируемая как пометка на дуге, связывающей данную функцию (операторную вершину) с ее параметром;

На уровне ОВМ описываемое таким образом отношение преобразуется в фрагмент ОВМ-сетн.

отношение:  $\text{пг} + \text{пг}$   
число:  $\text{число}$  число:  $\text{число}$

3 rminus#3.2\*1.2 \*2.1

Данное описание вводит отношение с именем  $pg+pg$ , определенное над тремя недоопределенными числами, слоты которых связаны между собой шестью функ-

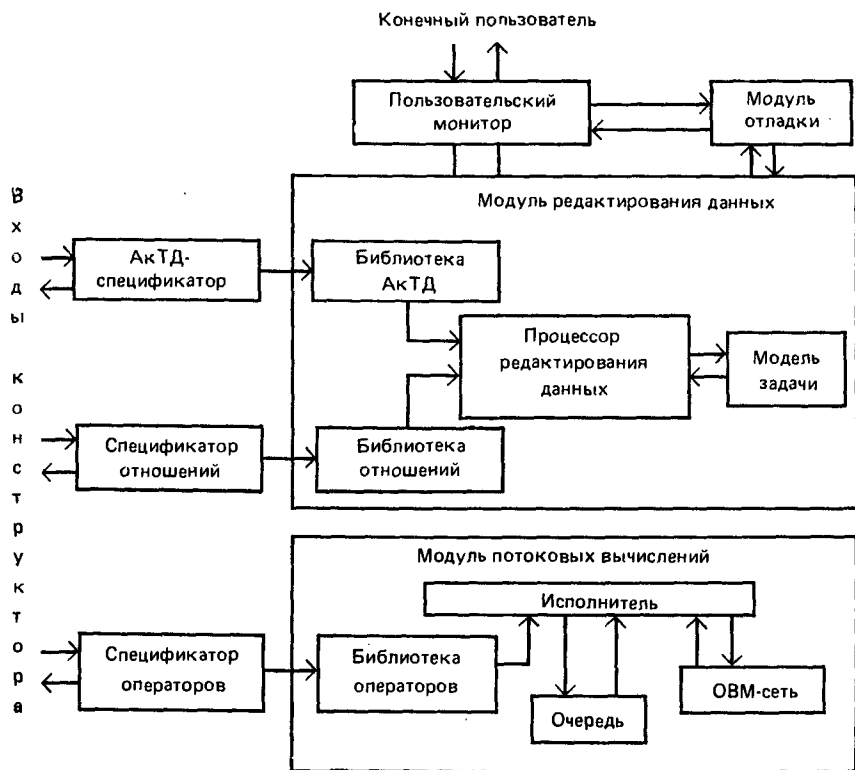


Рис. 3.2. Общая структура ТК-ОВМ

циями. Так, первая строка из раздела функций вводит функцию `grpus` с третьим приоритетом, аргументами которой являются значения первого слота ( $n$ -граница) второго и третьего аргументов отношения, а значением — первый слот его первого аргумента. Метки при параметрах функции говорят о том, что данная функция активируется при изменении любого из своих аргументов, а запись вырабатываемого значения производится через функцию присваивания. Поле подотношений отсутствует.

Средства спецификации АкТД и отношений позволяют пользователю формировать иерархически организованные системы понятий, описывающие достаточно сложные предметные области.

**Архитектура ТК-ОВМ.** Общая схема ТК-ОВМ приведена на рис. 3.2. Пользовательский монитор, модуль редактирования данных, модуль потоковых вычислений в совокупности образуют схему типового ОВМ-процессора. Модуль потоковых вычислений осуществляет исполнение ОВМ. Он включает лишь одну специфицируемую часть — библиотеку операторов (точнее, реализующих их процедур), наполнение которой определяет набор операторов (функций и предикатов), допустимых для данного ОВМ-процессора. Модуль редактирования данных, используя содержание своих библиотек, переводит модель задачи, формируемой пользователем в терминах АкТД и отношений, во внутреннее представление (ОВМ-сеть), а также обеспечивает возможность ее редактирования. Работа пользователя с ОВМ-процессором осуществляется через пользовательский монитор в интерактивном или пакетном режиме. Понятийный состав его

входного языка определяется наполнением библиотек АКТД и отношений. Допустимо формирование двух конфигураций ОВМ-процессора — полной (совпадающей с типовой) и базовой, в которой в отличие от первой отсутствует библиотека АКТД. Модель задачи для такого процессора формируется в терминах базовых типов данных МПВ и введенного в процессе проблемной настройки набора отношений.

ТК-ОВМ включает три подсистемы спецификации проблемного наполнения ОВМ-процессора: АКТД-спецификатор, спецификатор отношений и спецификатор операторов. Каждая из них имеет собственную архивную подсистему и транслятор внешних описаний в некоторое внутреннее представление, используемое в составе ОВМ-процессора. Работа с АКТД-спецификатором осуществляется в интерактивном режиме, который включает развитую систему подсказок и синтаксических шаблонов. Спецификаторы отношений и операторов допускают как пакетный, так и интерактивный режим работы. Пакетный режим позволяет наряду с ОВМ-процессорами производить системы, в которых данные средства настройки остаются доступными пользователю (как правило, через специальные интерфейсные оболочки). Кроме того, ТК-ОВМ-включает специализированный модуль отладки, который позволяет осуществлять «прокрутку» процесса исполнения ОВМ с управляемым шагом прерываний.

**Примеры использования ТК-ОВМ.** Использование ОВМ-процессоров в качестве ядра интеллектуальных систем оправдало себя при решении достаточно широкого круга задач в самых разнообразных предметных областях. На базе ТК-ОВМ реализовано более десятка демонстрационных интеллектуальных систем. В том числе «фабрика» по производству базовых ОВМ-процессоров с типовыми интерфейсными оболочками (типа «табло»), предоставляющая пользователю удобные средства для спецификации табличных и аналитически определяемых операторов, а также произвольных составных отношений. Здесь приведем краткую функциональную спецификацию лишь для трех из них: системы для согласования недоопределенных графиков работ (СНЕГ), системы событийного моделирования электронных схем (СИМОН) и автоматического решателя буквенно-цифровых головоломок (АРБУЗ). Выбор именно этих систем обусловлен отнюдь не их особым местом в ряду созданных на основе ТК-ОВМ, а лишь желанием показать разнообразие возможных применений данного комплекса.

Система СНЕГ реализована как комплекс вида ОВМ-процессор плюс база данных и поддерживает следующие режимы работы:

- ввод и редактирование предварительного графика работ с указанием различного рода причинно-временных характеристик и соотношений, в том числе недоопределенных;

- расчет календарного графика работ с учетом введенных соотношений. При этом система позволяет работать с приблизительными сроками выполнения и длительностями работ;

- поиск информации по реляционному запросу в предварительном или результирующем графике;

- утверждение рассчитанного графика работ или возврат к предыдущему варианту.

Наличие специализированной интерфейсной оболочки, предоставляющей пользователю проблемно-ориентированную многооконную систему для ввода, редактирования и визуализации данных, делают работу с системой простой и удобной. Так, функциональные экраны, имеющие форму карточек, заполняемых на работу и исполнителя/ресурс, поддерживают ввод информации, могут использоваться для формулировки запросов. Два других экрана служат для наглядной визуализации результатов расчета сетевого графика в виде таблицы, включающей предварительный и рассчитанный варианты, и в виде пары временных диаграмм. При формировании проблемного наполнения системы были использованы комплекс недоопределенных типов данных и формальная модель времени [Кандрашина, 1986].

Система СИМОН, явно использующая потоковую схему исполнения ОВМ, представляет собой экспериментальную реализацию одного из основных компо-

нентов компилятора — подсистемы событийного моделирования электронных схем (в данной версии системы в качестве базиса моделирования использована трехзначная логика, где третье значение соответствует неопределенному сигналу). Данная система предоставляет пользователю средства для определения библиотеки базовых элементов, моделируемой схемы, контрольных точек в ней и набора тестовых последовательностей. Собственно моделирование осуществляется ОВМ-процессором. Специализированная подсистема визуализации позволяет провести анализ результатов моделирования, сравнивая тестовые последовательности и динамику прохождения сигналов через контрольные точки, изображаемых в виде временных диаграмм.

Система АРБУЗ носит игровой характер, однако, безусловно, заслуживает упоминания ввиду нетипичности решаемых ею задач. Она ориентирована на решение буквенно-цифровых головоломок типа (SEND+MORE)=MONEY. Здесь каждая буква определяет одну цифру, разные буквы определяют разные цифры, самые левые буквы не должны быть равными нулю. Задача имеет однозначное решение, но в случае менее удачной постановки их может быть несколько, при этом пользователю предоставляется возможность задать дополнительные ограничения, сужающие значения букв, или перейти к новой задаче.

### **3.2. ПРИЗ — семейство инструментальных средств, ориентированных на знания**

*А. Калья, М. Коов, М. Кыпп, М. Мацкин, Я. Пеньям,  
Х. Перкманн, Э. Тыугу, Х. Хаав, А. Шмундак*

В ИК АН ЭССР разработано несколько ПРИЗ-ориентированных прикладных интеллектуальных систем. Все эти системы создавались на базе практически одной инструментальной среды — системы ПРИЗ, хотя, конечно, и сам инструментарий менялся по мере освоения новых идей, методов и программно-аппаратных средств. Сейчас уже можно говорить о семействе систем ПРИЗ и практическом использовании различных его представителей. Первоначально система ПРИЗ была реализована на ЭВМ типа «Минск-32», но в полном объеме она была развернута на ЕС ЭВМ [Кахро и др., 1981]. По мере совершенствования аппаратуры совершенствовалась и система. В нее включались новые технологические блоки, и на современном этапе можно говорить о втором поколении инструментальных ПРИЗ-систем.

#### **Система программирования ПРИЗ для ЕС ЭВМ**

Система ПРИЗ является модульной системой программирования, предназначена для построения пакетов прикладных программ, работает на всех моделях ЕС ЭВМ, начиная с моделей ЕС-1020 под управлением ОС 4.1 и выше, и полностью совместима с системами программирования Фортран, Ассемблер, Кобол и ПЛ/1.

Основным свойством системы ПРИЗ является автоматический синтез программ, что позволяет программистам создавать высококачественные ориентированные на пользователя пакеты прикладных программ. Семантическая модель предметной области описывается с помощью входного языка УТОПИСТ [Кахро и др., 1981]. При описании ориентированного на пользователя языка пакета можно использовать развитые макросредства. Разработанные на базе системы ПРИЗ пакеты прикладных программ полностью совместимы, что является необходимой предпосылкой создания интегрированных систем решения задач для разных предметных областей. Система ПРИЗ позволяет автоматизировать процесс построения крупных программ из заранее запрограммированных модулей, содержит средства отладки, обеспечивающие автоматизацию комплексной отладки синтезированных программ.

Система ПРИЗ имеет пользователей двух уровней: специалистов-практиков, использующих созданные пакеты прикладных программ по заданной проблеме для решения задач в повседневной работе, от которых требуется лишь знание входного языка пакета, и разработчиков пакетов прикладных программ, использующих систему ПРИЗ для разработки новых пакетов. Исходя из этого можно определить технологично использование системы ПРИЗ для построения пакетов прикладных программ как последовательность следующих основных этапов.

1. Описание и отладка моделей предметной области и связанное с этим конструирование входного языка пакета. Для этого используются в основном средства входного языка и директивы системы ПРИЗ. На этом этапе определяется, какие понадобятся понятия и какими параметрами они характеризуются. Практически определяется, какие задачи можно решать с помощью этого пакета. Так как вначале трудно, а иногда просто невозможно определить все необходимое, можно начинать работу с проектирования прототипа пакета. При имитации работы с пакетом на этом прототипе можно обнаружить недостатки, которые не разрешаются полностью реализовать возможности пакета. Обычно на этом этапе появляются и новые требования к пакету. После анализа работы прототипа пакета составляется описание в полном объеме.

2. Программирование и автономная отладка отдельных программных модулей, написанных на других языках программирования.

3. Обеспечение услуг, необходимых для работы с пакетом, например, в пакетном или диалоговом режиме, связь с данными, полученными в других программах, удобный ввод-вывод.

Таким образом, система ПРИЗ-ЕС является достаточно мощной технологической системой разработки пакетов прикладных программ. Вместе с тем появление микроЭВМ потребовало модификации системы ПРИЗ. Обсуждению одной из них и посвящен следующий подраздел.

### **Системы МИКРОПРИЗ и МИКРОЭКСПЕРТ**

Системы МИКРОПРИЗ и МИКРОЭКСПЕРТ позволяют разработчикам прикладных интеллектуальных систем, с одной стороны, использовать концептуальные знания в виде всегда справедливых в определенной предметной области законов, а с другой — вкладывать в систему экспертные знания. Этот симбиоз дает пользователям мощное средство реализации прикладных систем в таких, например, областях, как САПР, где формальные модели активно используются совместно с логическими рассуждениями.

Желаемое поведение прикладной системы определяется на основе предъявлении примеров правильного поведения в типичных случаях и спецификации конкретных вычислительных задач.

Обе системы разработаны в ИК АН ЭССР, написаны на языке Паскаль и работают на микроЭВМ с операционными системами MS DOS, CP/M-86 или CONCURRENT-DOS.

Система МИКРОПРИЗ. Являясь интеллектуальной системой с автоматическим синтезом программ, система позволяет пользователю быстро и эффективно формировать собственный набор понятий, а затем использовать его для описания разных прикладных задач. По этим спецификациям система автоматически находит способ решения прикладной задачи, синтезирует соответствующую программу и выполняет ее, работает с концептуальными знаниями. Система реализована как интерактивная система программирования для микроЭВМ с входным языком высокого уровня, близким к языку УТОПИСТ [Кахро и др., 1981; Тыгуу, 1984].

В него включены все основные возможности языка УТОПИСТ с учетом специфики интерактивной работы и реализации языка на микроЭВМ. Главное отличие здесь — более четкая граница между абстрактными и конкретными объектами. И в этом смысле входной язык МИКРОПРИЗа ближе к традиционным объектно-ориентированным языкам. В нем разделены описание абстрактных объектов, соответствующих классам, и порождение конкретных объектов, участ-



вующих в вычислениях. Условно входной язык систем делится на две части: язык решения задач и язык описания понятий. Эти части синтаксически аналогичны (за исключением того, что в одной из них отсутствует предложение постановки задачи), но различны по семантике.

Язык решения задач предназначен для описания вычислительной модели задачи и собственно постановки задачи. Модель задачи состоит из объектов, каждый из которых принадлежит определенному классу, и отношений между объектами. Модель задачи наращивается постепенно, порождение объектов и задание дополнительных отношений обычно чередуются с вычислениями. Предложение постановки задачи имеет два варианта. Первый предназначен для вычисления значений перечисленных в нем объектов, а второй — для поиска аналитических выражений для перечисленных объектов через заданные.

Предложения входного языка могут поступать в систему как с клавиатуры, так и из текстового файла. В последнем случае оказывается полезной возможность присваивать объектам значения в ходе вычислений. Для этого в системе МИКРОПРИЗ существует специальная команда, после ввода которой значение соответствующего объекта запрашивается у пользователя.

Каждый объект в модели задачи принадлежит к некоторому классу. Кроме системных (встроенных) классов, таких, как число, текст, неопределенный, структура, существует потенциально бесконечное число классов, определяемых самим пользователем. Их описания хранятся в системной базе знаний и/или в базах знаний пользователей. Далее «пользовательские» классы будем называть понятиями, чтобы отличать их от системных классов. Понятие — совокупность знаний об объекте и/или явлении, представляемое в виде вычислительной модели. Понятие имеет компоненты, которые, подобно объектам, принадлежат некоторому классу. При порождении объекта по описанию понятия порождаются и все объекты, соответствующие компонентам понятия. Последние могут быть связаны между собой отношениями. Наряду с уравнениями и структурными отношениями допускаются (хотя и в ограниченном варианте) и отношения в виде программных модулей. Ограничения по сравнению с системой ПРИЗ-ЕС связаны в первую очередь с тем, что пользователями системы МИКРОПРИЗ являются непрограммирующие профессионалы. Поэтому здесь не допускается подключение программ, написанных на других языках программирования, хотя имеется совокупность системных модулей, частично компенсирующих этот недостаток. Описания понятий, содержащих такие модули (например, циклы, экстремумы функций, печать и т. п.), хранятся в системной базе знаний. Возможность использования их в составе других понятий и наличие в системе МИКРОПРИЗ режима подзадач позволяют гибко решать многие вычислительные задачи.

Описание понятий происходит в специальном режиме на языке, синтаксически совпадающем с языком решения задач. При возврате в режим решения задач система записывает полученное понятие в базу знаний. В дальнейшем оно может быть использовано при описании задач и новых понятий.

МИКРОПРИЗ — интерактивная система программирования с многооконным интерфейсом. Здесь имеется текстовый редактор для подготовки полного текста задачи. Но можно использовать и режим непосредственного ввода, в котором обработка каждой строки и ее выполнение происходит сразу. Правда, в этом режиме пользователь не может редактировать текст, введенный ранее. Управление работой системы осуществляется в режиме меню (рис. 3.3). Команды можно выполнять, выбирая их из меню или включая в текст, выполняемый в режиме решения задачи.

Типичный сеанс работы с системой МИКРОПРИЗ описывается следующим образом. Вначале система находится в режиме текстового редактора. Пользователь готовит текст своей задачи или описания нужных ему новых понятий на входном языке системы, оперируя понятиями базы знаний из своей предметной области, затем выбирают из подменю Execute команду для выполнения подготовленного текста в режиме решения задач или описания понятий. В зависимости от режима система либо создает вычислительную модель и проводит на ней соответствующие вычисления, либо записывает новое понятие в базу знаний.

ESC to exit menu

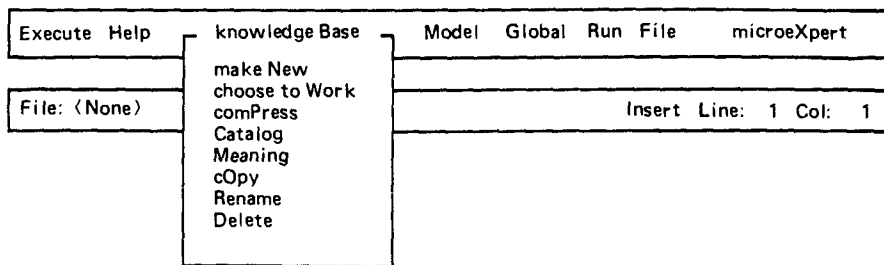


Рис. 3.3. Меню системы МИКРОПРИЗ

Затем пользователь редактирует тексты и обрабатывает их снова или проводит вычисления на модели с разными начальными данными. По окончании работы набранные тексты можно сохранить в файле, который можно заново обрабатывать, как бы превращая их в программы на входном языке системы.

Для обработки текстового файла в подменю Execute служит соответствующая команда. Команды в подменю Help предназначены для обеспечения пользователя вспомогательной информацией о работе с системой. Команды подменю knowledge Base позволяют создать новые базы знаний и выбрать подходящую базу для работы, посмотреть каталог базы знаний и описания понятий и т. д. Подменю Model содержит команды для работы с моделью задачи (просмотр модели, присваивание значений объектам, удаление старой модели). Подменю Global предназначено для изменения параметров системы, в том числе точности вычислений, начальных точек для итераций при решении уравнений, формата выводимых чисел. Из этой же опции можно заказывать печать алгоритма решения задачи после каждой ее постановки. Из подменю Run можно запустить программы, встроенные в систему. Подменю File содержит команды текстового редактора для работы с файлами и команду Quit — выхода из системы. Последнее подменю — microexpert — относится к системе МИКРОЭКСПЕРТ.

В системной базе знаний находятся некоторые системные понятия и понятия, соответствующие простым геометрическим фигурам и элементам электрических цепей. При использовании этих понятий на экран выводятся их графические изображения с пояснениями. Пользователям предоставляется возможность создавать собственные графические комментарии с помощью графического пакета REGIS.

**Система МИКРОЭКСПЕРТ.** Выбор опции MICROEXPERT в основном меню запускает систему МИКРОЭКСПЕРТ, которая может работать как совместно с системой МИКРОПРИЗ, так и независимо. Система позволяет работать с базой знаний (накопление и изменение знаний) и запускать запросную систему ЭС.

Экспертные знания представляются в виде примеров из конкретной области, в которых определяются условия и результаты вычислений, принятия соответствующего решения или вывода значения вычисленных переменных. Для использования системы МИКРОЭКСПЕРТ совместно с системой МИКРОПРИЗ в экспертные знания включаются еще спецификации вычислительных задач на входном языке МИКРОПРИЗа. Примеры и спецификации вычислений на входном языке МИКРОПРИЗа образуют экспертную базу знаний. На основе заданных знаний МИКРОЭКСПЕРТ генерирует запросную систему, при запуске которой пользователю задаются вопросы, при необходимости выполняются нужные вычисления и, наконец, выводится результат, соответствующий выбранным ответам и результатам вычислений и сравнений.

Для построения экспертной базы знаний для конкретной проблемной области разработчик выделяет набор атрибутов, характеризующих заданную область, и примеры, причем каждому атрибуту соответствуют в примерах конкретные зна-

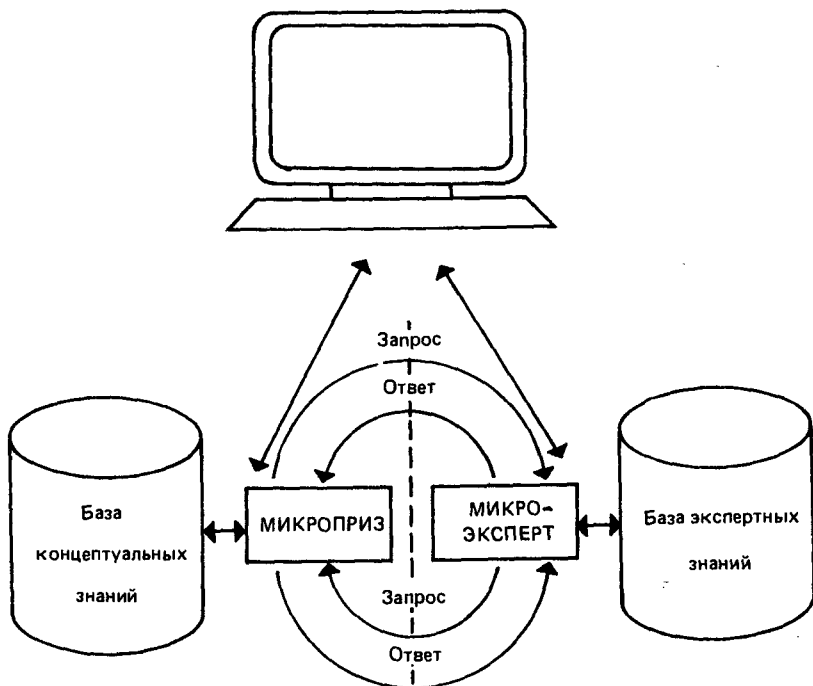


Рис. 3.4. Схема взаимодействия систем МИКРОПРИЗ и МИКРОЭКСПЕРТ

чения. Атрибуты определяют возможные действия, которые выполняются во время работы запросной системы: вывод вопроса, вычисление, сравнение значений вычислительных переменных, вывод результата. Вопросы задаются в форме меню, содержащего возможные ответы — значения из примеров, соответствующие атрибуту. Тексты запросов могут быть включены в знания о проблеме. При этом можно создавать проблемно-ориентированный язык соответствующей прикладной области. Если текст отсутствует, задается стандартный вопрос: какое значение имеет <имя атрибута>? Пользователь делает свой выбор, двигая курсором по возможным ответам. При атрибутах, связанных с вычислением, совершаются нужные вычисления по описаниям задач, запрашивает начальные значения у пользователя. Имеется возможность использования в примерах специального значения «не обращая внимания», которое означает, что в данном примере по соответствующему атрибуту не надо выполнять никаких действий.

При решении задач, которые требуют совместного использования систем МИКРОПРИЗ и МИКРОЭКСПЕРТ, они взаимодействуют в объектно-ориентированном стиле, посылая друг другу сообщения (рис. 3.4). Связь между системами организована таким образом, что систему МИКРОЭКСПЕРТ можно вызывать рекурсивно. Возможны два режима взаимодействия между этими системами:

1. По накопленным экспертным знаниям и ответам на вопросы, заданные пользователем в ходе диалога с экспертной системой, происходит синтез спецификации задачи, выбор вычислительной модели или принятие решения. При этом варианте пользователь запускает систему МИКРОЭКСПЕРТ специальной командой системы МИКРОПРИЗа из подменю и при необходимости происходит автоматическое переключение на МИКРОПРИЗ для выполнения нужных вычислений.

По завершении работы пользователь может проверить цепочку рассуждений по выводу решения, запуская подсистему объяснений.

2. По спецификации задачи, полученной с использованием концептуальных знаний на уровне МИКРОПРИЗа, выполняется синтез программы и во время вычислений происходит обращение к экспертным знаниям. В базе знаний МИКРОПРИЗа существует класс (понятие) Эксперт, соответствующий «абстрактному» эксперту, способному помочь при решении задач. Класс Эксперт содержит имя базы знаний и результат, а также отношение между ними, вычисляющее результат по имени базы знаний. При порождении в вычислительной модели задачи конкретного объекта класса Эксперт уточняют область применения, задавая имя соответствующей экспертной базы знаний. Результат или ответ эксперта может быть связан со значениями других объектов, т. е. использован в вычислениях. Отношение между областью применения и ответа эксперта реализовано программой — ядром системы МИКРОЭКСПЕРТ. Это значит, что выполнение синтезированной МИКРОПРИЗом программы для решения задачи включает диалог с пользователем, цель которого — выяснить экспертную оценку нужных параметров заданной проблемы. В этом режиме МИКРОПРИЗ запускает систему МИКРОЭКСПЕРТ при выполнении синтезированной программы. Такой подход целесообразно использовать прежде всего для определения эмпирических значений параметров, участвующих в вычислениях.

**Примеры использования систем МИКРОПРИЗ и МИКРОЭКСПЕРТ.** Система программирования МИКРОПРИЗ оснащена диалоговыми пакетами прикладных программ, в которых применяется синтез программ как основа процесса решения задач. Структура вычислительного процесса во всех пакетах определяется модельной базой знаний о предметной области. Каждый пакет имеет свою базу знаний в виде отдельного файла. В системе МИКРОПРИЗ разработаны следующие пакеты:

- система базы данных МИКРОДБ,
- пакет проектирования валов в сборе,
- пакет проектирования зубчатых колес,
- пакет контрольных расчетов зубчатых колес,
- пакет проектирования радиотехнических активных фильтров,
- пакет моделирования динамических систем,
- пакет расчета линейных цепей переменного тока,
- пакет машинной графики,
- пакеты школьной информатики.

Этот список говорит о том, что система МИКРОПРИЗ особенно хорошо подходит для реализации пакетов программ инженерных расчетов. Набор пакетов выбран так, чтобы их можно было реализовать и в новой системе программирования НУТ. Ниже мы рассмотрим два пакета. Система базы данных предназначена для обработки данных и может быть использована в других пакетах как средство хранения информации. Пакет проектирования валов в сборе — характерный пакет инженерных расчетов. Он включает все основные средства программирования системы МИКРОПРИЗ и применяет средства МИКРОЭКСПЕРТА.

Система базы данных МИКРОДБ обеспечивает пользователя следующим подмножеством операций СУБД реляционного типа: выборка на основе логического условия; проекция отношения; создание нового отношения по определенной структуре кортежа. При этом атрибуты кортежа могут быть получены прямо из исходного отношения, вычислены по уравнениям как результаты выполнения некоторого пакета программ.

Отношения реализованы как плоские файлы. Кортежу отношения соответствует запись файла, а атрибуту — поле данных. Обслуживание — ввод, редактирование — базы данных осуществляется с помощью отдельных сервисных программ, которым соответствуют команды из подменю Run системы МИКРОПРИЗ.

В системе МИКРОДБ реализованы два способа ввода, редактирования и вывода данных: в виде таблиц и макетов ввода-вывода. В случае использования

табличного способа ввод и редактирование исходных таблиц (файлов) осуществляются редактором текстов системы МИКРОПРИЗ. В случае выборки подмножества результат выводится на экран в виде такой же таблицы, как и в исходном файле. Если запрос включает создание нового файла, то от пользователя запрашивается заголовок новой таблицы, по которому выводится результат. При макетном обмене вначале создается макет ввода-вывода, который потом используется для ввода, редактирования или вывода данных. Макет подготавливается с помощью редактора текстов системы МИКРОПРИЗ как стандартный текстовый файл, который состоит из комментариев и полей данных. Поле данных и его длина указываются звездочками (\*) в соответствующем месте макета:

```
infile — имя входного файла,
inрес — структура записи входного файла,
cond — логическое условие,
outrec — структура выходного файла,
outfile — имя выходного файла,
format — имя файла формата вывода,
res — результат выполнения запроса при табличном выводе результата,
fres — результат выполнения запроса при макетном выводе результата.
```

Доступ к базе данных организован на уровне входного языка системы МИКРОПРИЗ. Чтобы иметь доступ к базе данных, необходимо породить конкретное понятие SETS и определить его компоненты.

Например, для получения результата надо указать в постановке задачи компонент res или fres. Если запрос данных включает требование к созданию нового файла, надо определить компоненты inрес и outrec, имена исходного и выходного файлов, а также записи исходного и выходного файлов. Элементы записи выходного файла могут быть вычислены по уравнениям или получены из исходного файла, могут быть и результатами работы другого пакета программ:

```

} person1 (name fname dep prof salary year)
} person2 (name fname salary tax income)
} inc : SETS infile='empl.db' inрес=person1
      outrec=person2 outfile='income.db'
}
} tax=salary*0.08
} income=salary—tax
} ? inc.res
```

Здесь результат выводится в виде таблицы. Если дополнительно указать, например, что компонент format='PIC.DB', и потребовать вычислить inc.res, то результат будет выводиться поэлементно (по записям) на основе макета, который находится в файле PIC.DB.

Система базы данных МИКРОДБ имеет оконный интерфейс для всех своих сервисных средств и для вывода результатов запроса. Кроме результатов в окнах отражается и полезная для пользователя вспомогательная информация о текущих файлах базы данных, атрибутах и др.

### Система программирования НУТ

Система программирования НУТ основана на объектно-ориентированном языке НУТ [Тыгу и др., 1985]. Являясь ядром прикладных систем, система НУТ предоставляет широкий спектр услуг — от возможностей простого калькулятора до средств системного программирования. Но главное, что определяет применимость системы НУТ, это простота ее расширения путем создания новых пакетов и перегруппирования знаний в существующих пакетах. В систему включен синтезатор программ, обеспечивающий эффективное использование большого объема знаний о разных предметных областях. Благодаря многооконному графическому интерфейсу общение с системой просто и удобно. Система НУТ реализована на языке Си для операционных систем CP/M-86 и UNIX для ПЭВМ с 16- и 32-рядными микропроцессорами и требует не менее 500К оперативной памяти.

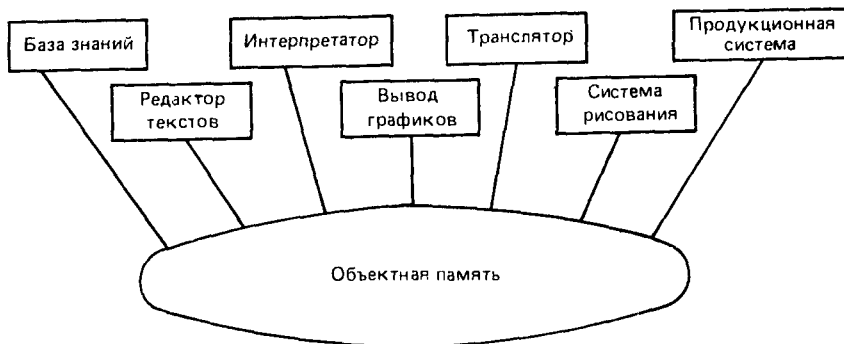


Рис. 3.5. Общая структура системы НУТ

Общая структура системы НУТ приведена на рис. 3.5, где перечислены основные компоненты, их взаимосвязь через универсальное внутреннее представление системных наборов данных (объектную память) и связь с многооконным интерфейсом.

В пакетах базы знаний накапливаются как абстрактные универсальные знания, так и знания о некоторой предметной области (например, знания о задачах, связанных с конкретным проектируемым устройством, о задачах моделирования определенного района). Каждый пакет относится к некоторой предметной области и носит ее имя, например геометрия, физика, САПРСАУ. Пакет содержит классы, объекты и правила. Классы являются описаниями понятий, например, в геометрии это треугольник, отрезок, точка. Объекты представляют собой составные величины, соответствующие реальным сущностям и обязательно принадлежащие определенному классу. Например, вода — это объект класса жидкость, имеющий конкретные значения характеристик точка кипения, удельная вязкость и т. д. Эти характеристики являются компонентами объекта. Наконец, в пакет входят правила, выражающие законы общего вида, справедливые для предметной области пакета. При создании пакета можно копировать в него классы, объекты и правила из уже имеющихся пакетов. По мере уточнения знаний о предметной области пакет можно редактировать. Для создания и редактирования пакетов используется многооконный интерфейс.

В отличие от традиционного (телетайпного) интерфейса, при котором на экране дисплея в хронологической последовательности отображается взаимодействие пользователя с системой, многооконный интерфейс позволяет располагать информацию на экране наиболее удобным для пользователя способом, быстро переключаться с одного вида деятельности на другой и, наконец, перейти от «операционного» стиля взаимодействия, когда сначала указывается оператор, который нужно выполнить, а затем его операнд(ы), к более естественному объектно-ориентированному, когда сначала выбирается объект, а затем из меню выбирается нужное действие.

В системе НУТ используется многооконный интерфейс с перекрывающимися окнами. В окнах располагается информация о состоянии системы: в одном — список имеющихся в системе пакетов, в другом — список классов в некотором пакете, в третьем — описание некоторого класса и т. д. Окон каждого типа может быть несколько. Если окно перестает быть нужным, его можно свернуть, тогда на экране останется так называемая бирка, используя которую можно впоследствии снова развернуть окно. Верхняя строка экрана выделена для меню. В ней содержатся команды, связанные с текущим активным окном.

В каждый момент времени пользователь работает только с одним окном (оно называется активным и видно на экране целиком). Для перехода к другому окну

(например, если мы хотим перейти от описания класса к пакету целиком) нужно, пользуясь настольным манипулятором («колобком»), подвести курсор к окну пакета и нажать кнопку на колобке.

Язык НУТ является развитием языка УТОПИСТ (НУТ — Новый УТОПИСТ) [Кахро и др., 1981]. Принципиальное отличие нового языка от УТОПИСТА заключается в его динамизме — он рассчитан на современную вычислительную технику, способную обеспечить достаточную эффективность реализации при динамической обработке типов. Второй особенностью языка НУТ является то, что программа решения задачи, готовая к исполнению, полностью скрыта от пользователя. Пользователь оперирует объектами, заставляя машину выполнять разные действия над ними, пользуется знаниями о свойствах объектов, в частности об их вычислимости, но, как правило, не знает, какие программы должны вызываться для выполнения заказанных действий. Например, выполнение оператора  $y := m.f(x)$  может потребовать автоматического синтеза программ, если функция  $f$  задана отношением с подзадачами. Третье существенное изменение в языке — явное введение операций над объектами (введение мощности процедурной части). Наконец, новыми в языке являются продукция, позволяющие задавать общеприменимые знания, а также описывать задачи в стиле языка Пролог там, где это допустимо по эффективности.

Единицей знаний, задаваемых на языке НУТ, является класс — носитель свойств объекта. Класс определяет структуру объектов (экземпляров) и набор допустимых операций. Операции описываются через отношения. Кроме того, возможно указание инициализации компонентов и виртуальных компонентов (т. е. таких, которые используются при вычислении, но не участвуют в формировании значения объекта описываемого класса). В системе имеются встроенные классы. Свойства объектов этих классов определены системой в виде операций, которые можно выполнять над ними. Остальные классы определяются пользователем.

В простейшем случае явного описания класса задаются только компоненты. Например,

```
point: (x:numeric;  
        y:numeric);
```

определяет класс point объектов с числовыми компонентами  $x$  и  $y$ . Еще одним примером явного описания класса может служить описание множества:

```
set: (value: any;  
      vir elem: any;  
      key: any;  
      selector: numeric;  
      r1: value, selector → elem [c-fun A];  
      r2: value, key → elem [c-fun B];  
      put: value, elem → value [c-fun C]);
```

Значением компонента value является множество целиком. Виртуальные компоненты elem, key, selector представляют текущий элемент, ключ и признак, задающий способ выборки элемента. Определены отношения выборки элемента по ключу r2 и по признаку selector(r1). Эти отношения заданы программами B и A. Третье отношение put, заданное программой C, вычисляет новое значение value по заданным value и elem.

При описании компонентов класса можно пользоваться не только встроенными, но и любыми ранее описанными классами. Например, описатель

```
PO: point;
```

содержит имя ранее созданного класса. Кроме имени класса можно указывать изменения свойств класса. Например,

```
Unary: comp mod=1;
```

фиксирует значения компонентов. Но возможны и более сложные изменения, например

```
Points: set elem → PO;
```

В данном случае  $P0$  определяет класс, объектами которого будут значения компонента `elem` объекта `points`. Заметим, что  $P0$ , `Uparg` и `Points` — не новые классы, а компоненты класса, внутри которого они находятся. Компоненты класса определяют поименованные объекты, которые будут созданы при порождении объекта данного класса. При описании новых классов возможно наследование свойств ранее описанных классов с помощью конструкции `super`.

Все вычисления в системе организуются на уровне объектов. Значение объекта может быть простым или составным. Элементом составного значения объекта может быть значение другого объекта. Таким образом, составной объект может иметь произвольную иерархическую структуру, однако доступ через имена возможен только к непосредственным компонентам. Если отсутствует инициализация объекта или это специально не сказано при порождении объекта, то его начальное значение и все элементы составного значения считаются равными `nil`. Существуют предопределенные объекты (константы). Это числа, текстовые константы, булевские значения `true` и `false`, некоторые системные объекты. Объекты таких типов называются первичными.

Новые объекты порождаются в ходе вычислений в результате выполнения функций. Имеется специальная функция `new`, вырабатывающая новый объект данного класса. Например, вызовом функции

```
z := new point;
```

создается новый объект `z` класса `point`. При этом для него выделяется память, а его компонентам `x` и `y` присваивается значение `nil`.

Программы, являясь объектами, могут порождаться как явно, так и неявно. Уравнение задают программы неявно. Например `x+y=z`; порождает три объекта класса `prog` со значениями

```
x := z-y;
y := z-x;
z := x+y;
```

Программа `max1` — вычисления максимального значения функции от одного аргумента (`amax`, `frax`) на заданном интервале (`from`, `to`, `step`) — пример явного порождения программы:

```
max1 := new prog (fun, from, to, step→amax, fmax)
{
    fun (from, fmax);
    amax := from;
    for i=from+step step-step to-to
    do
        fun (i, f);
        if f<fmax→fmax := f; amax := i fi
    od
}
```

Конструкция (`fun`, `from`, `to`, `step→max`, `frax`) определяет тип программы. Данная программа пользуется параметром `fun`, имеющим класс `prog`. Последовательность операторов, обрамленная в операторные скобки `begin` и `end`, является телом программы.

Использование объекта `max1` возможно двумя путями:

1. Если создан объект `fun` класса `prog`:

```
fun := new prog (x→y)
{
    y = -(x~2)
}
```

то в рабочей области можно написать

```
max1 (fun, -1, 1, 0.1, arg, f);
```



2. Можно ссылаться на объект как на реализацию некоторого отношения, например

```
max :=  
(arg→fun), beg, end, step→argmax, funmax{prog max1};
```

Для вычислений можно также непосредственно применять правила. Правила имеют форму импликаций и выражают факты и законы, по которым можно устанавливать новые факты (строить новые описания, создавать новые объекты). Правила-факты, например, следующие:

```
→отец (Иван, Петр)  
→отец (Сергей, Иван),
```

а правило-закон

отец (@x, @y) & отец (@y, @z) → дед (@x, @z). В законы входят переменные — идентификаторы, начинающиеся с символа @. По заданному закону из приведенных фактов сразу выводится новый факт:

```
→дед (Сергей, Петр).
```

Природа предикатов отец и дед может быть разная. Они могут быть абстрактными символами, именами программ, которые можно выполнить, именами классов. Последнее очень полезно для описания общих законов, действующих над объектами ранее определенных классов.

Проиллюстрируем возможности языка НУТ, приближающие его к языку Пролог. Опишем законы, связывающие расстояния между точками на оси, предположив, что расстояния могут измеряться в разных единицах: метрах, сантиметрах и миллиметрах. Пусть описания классов Длина и Дист(анция) следующие:

```
Длина: (m: numeric; %*длина в метрах %  
vir SM, MM: numeric;  
SM=100*M;  
MM=10*SM);  
Дист: (A, B: точка;  
L: Длина);
```

Согласно первой строке объект класса Длина имеет значением длину в метрах. Вторая строка, начинающаяся со слова vir, задает виртуальные компоненты объекта, длину в сантиметрах и в миллиметрах, которые могут использоваться в вычислениях наравне с метрами. Однако любой объект x, описанный как

x : Длина;

будет иметь значение только в метрах. В частности, описания

```
x1 : Длина 5;  
x2 : Длина см=500;
```

задают одно и то же. По ним можно вычислить значение и в миллиметрах, которое равно 5000.

В языке предусмотрено автоматическое приведение составного значения объекта к первичному, если составной объект имеет единственный компонент и этот компонент первичного типа. Благодаря этому объекты класса Длина можно использовать в уравнениях наравне с числовыми, например объекту x можно задать значение 10 (метров) уравнением  $x = x_1 + x_2$ . Закон о сумме расстояний задается следующим правилом-законом:

```
Дист (@u, @x, @y, @L1) & Дист (@v, @y, @z, @L2)  
→ Дист (#, @x, z, @L1 + @L2);
```

Реализацию некоторых функций языка Лисп в системе НУТ покажем на следующем примере. Сначала описывается класс list:

```

var car, cdr: any;
далее класс Lisp:
var l, li, l2: any;
rel
  car: li → r {r := li.car};
  cdr: li → r {r := li.cdr};
  cons: l1, l2 → r {r := new list; r.car := l1; r.cdr := l2};
  nil: li → r {r := nil};
  append: l1, l2 → r {
    if l1 == nil → r := l2
    true → r := l.cons (l1.car, l.append (l1.cdr, l2))
  };
  reverse: l1, l2 → r {
    if l1 == nil → r := l2
    true → r := l.reverse (l1.cdr, l.cons (l1.car, l2))
  };

```

Использовать описание классов можно, например, следующим образом:

```

l := new Lisp;
L1 := l.reverse (l.append (l.cons (l.cons (2, nil)),
  l.cons (4, l.cons (5, nil))));

```

В результате исполнения этой последовательности действий объект L1 получит значение (5 4 2 1).

Задача синтеза программ в системе НУТ активируется при вызове функции  
M.COMPUTE (x1, ..., xn),

где x1, ..., xn — компоненты объекта M, значения которых должны быть вычислены при исполнении данного вызова либо при вызове отношения, реализация которого не указана.

Поскольку такой вызов может стоять в выражении в теле цикла, а его обработка зависит от наличия значений компонентов объекта M, планирование вычислений и синтез программ выполняется в общем случае динамически, при каждом исполнении этого цикла. Это крайний случай динамизма. Другой крайностью является ситуация, когда заранее, в ходе трансляции текста программы, можно выполнить синтез и скомпилировать эффективный код. Для этого должно быть известно, что контекст, в котором выполняются вычисления, не изменится. Точнее, класс объекта M и определенность значений компонентов этого объекта сохраняются во время вычислений. В этом смысле задача компиляции здесь превращается в задачу оптимизации, учитывающую постоянство контекста. В связи с этим в системе НУТ реализуются оптимизаторы (компиляторы) разной степени сложности, выполняющие смешанные вычисления над программой и частично зафиксированным контекстом.

Выше мы рассмотрели основные возможности системы НУТ, разработанной для профессиональных ПЭВМ. В настоящее время система используется для разработки пакетов прикладных программ. В ней реализованы пакеты, уже обсуждавшиеся при рассмотрении систем МИКРОПРИЗ и МИКРОЭКСПЕРТ.

### 3.3. Инструментальный технологический комплекс для систем семиотического моделирования

*А. А. Абдрахманов, В. С. Лозовский, С. В. Лозовский*

Инструментально-технологический комплекс — базовая интеллектуальная система (ИТК-БИС) — служит для повышения эффективности процесса разработки систем семиотического моделирования и управления, а также для создания

операционной среды применения подобных систем, отвечающей требованиям промышленной эксплуатации. При его создании были поставлены следующие задачи.

1. Выбрать инструментальный язык для реализации систем семантического моделирования достаточно высокого уровня, позволяющий легко работать со сложными динамическими структурами данных (в частности, деревьями и графами), обладающий простым и единообразным синтаксисом, предоставляющий удобные и лаконичные средства для создания стратифицированных архитектур языковых надстроек, реализации специфических структур данных и принципов управления процессом вычислений.

2. Выбрать тип и структуру базовой системы представления данных и знаний (СПДЗ), обладающей достаточно мощными и выразительными средствами построения семантических моделей для управления сложными процессами, ориентированными на формализацию знаний широкого спектра предметных областей, получаемых на всех этапах исследования, включая анализ технической и эксплуатационной документации, а также знаний отдельных специалистов-экспертов. Непременным требованием к языку СПДЗ, базовым и инструментальным средствам работы с нею должна быть простота модификации в процессе развития и, как следствие, постоянно меняющихся условий и требований к режимам функционирования.

3. Комплекс должен включать единообразные простые средства интеграции существующих программных модулей и пакетов. Особую важность это требование приобретает на этапах внедрения, когда разрабатываемые средства создаются в условиях наличия у конечного пользователя обширного задела по решению отдельных функциональных задач. Услугами разработчиков пакетов воспользоваться зачастую не представляется возможным; их существенная модификация, как правило, невыполнима, поэтому процесс включения в общую интегрированную структуру нового программного обеспечения (ПО) должен выполняться максимально простыми и лаконичными средствами.

4. Системы семантического моделирования и управления, как правило, представляют собой одну из подсистем в сложной структуре взаимодействующих разнородных программных комплексов. Поэтому на общественном уровне необходимы универсальные, простые, надежные и лаконичные средства межподсистемного интерфейса, позволяющие с максимальной эффективностью на определенных технических средствах обеспечивать функционирование требуемого числа подсистем, их запуск, останов и удобный обмен информацией в нужной форме. Недооценка этого требования приводит к тому, что разработанные новые средства никогда не будут использоваться на практике, как бы хорошо они ни проявили себя на этапе автономного функционирования и при решении демонстрационных задач.

5. Целевые программные комплексы автоматизированного управления, частью которых являются создаваемые средства новых информационных технологий, сами постоянно усложняются. Управление их функционированием становится нетривиальной задачей. Особенно усложняется управление такими программными комплексами, когда приходится принимать оперативные решения по сохранению наиболее важных результатов, процессов, по переводу комплекса в безопасное состояние с целью предупреждения развития катастрофических последствий. Сложность всех этих требований, недопустимость на данном этапе ошибок и жесткие временные рамки требуют неизбежной автоматизации перечисленных функций. Положение усложняется еще и тем, что вся логика совместного управления чрезвычайно вариантна, в нее требуется постоянно вносить изменения, причем делает это оператор в процессе штатной эксплуатации, что исключает предпрограммирование.

6. Совокупность программных средств и возможностей по программной надежности, быстройдействию и расходу наиболее дорогостоящих ресурсов вычислительных установок должна соответствовать требованиям промышленной эксплуатации.

## Структура ИТК-БИС

Исходя из перечисленных требований и был разработан ИТК-БИС. Инструментальный комплекс представляет собой концентрическую систему уровней ПО:

МПО (V)	Совокупность информационных моделей ФПО
ФПО (IV)	Монитор SMS сценарного управления вычислительным процессом Другие функциональные подсистемы
БПО (III)	СПДЗ H-P/REX
ИПО (II)	Интерпретатор и компилятор ОЛИСП Пакеты FEND, STNDRD, DLGF, TABLC, STEND, GMSF, UNIS языкового расширения Метод доступа RLS с утилитой SERVRLS Пакет PPR форматной распечатки ОЛИСП-текстов
ОСПО (I)	Монитор GMS операционной интеграции задач Дисплейный модуль DMF Препроцессоры интерфейса PPMPL, PPMFORT Транслятор видеоформ FORMTRAN
ЕС ЭВМ	Ряд-2, ОС MVT, SVS, CBM ОЗУ (1 Мбайт)

Ее ядром является операционная система (ОС) ЕС MVT 6.1, SVS или CBM/БОС. Возможности ОС расширяются с помощью уровня общесистемного ПО (ОСПО). Это открытая подсистема интеграции отдельных функциональных подсистем, допускающая простое и легкое наращивание и модификацию. Архитектура ОСПО накладывает минимальные ограничения на логику и дисциплину функционирования отдельных подсистем, предусматривает их параллельное асинхронное функционирование, гибкие возможности взаимной синхронизации и обмена сообщениями, обеспечивает включение в архитектуру модулей на различных языках программирования; допускает организацию и поддержку гибких диалоговых режимов с использованием расширяемой номенклатуры терминальных устройств; в архитектуре предусмотрена защита отдельных подсистем от непредусмотренных перекрестных влияний.

Развитые операционные системы должны обеспечивать перечисленные возможности своими средствами. К сожалению, в ОС ЕС указанные функции непосредственно не реализуются, что потребовало соответствующих доработок ОСПО. Большое значение для удобства организации человеко-машинного диалога имеет разработка гибких и удобных средств формирования экранов дисплеев с целью вывода и ввода различных таблиц, сообщений, запросов и т. д. Поэтому были созданы язык FL [Лозовский С., 1985] и процессор видеоформ FORMTRAN. В ОСПО предусмотрены специальные средства включения в архитектуру ИТК-БИС программ и подсистем, написанных на Ассемблере, ПЛ/1, Фортране и Лиспе.

Инструментальное программное обеспечение (ИПО) включает основной языковый процессор выбранного языка реализации, пакеты, расширяющие его функции, и сервисные утилиты. Основу ИПО составляет система ОЛИСП [Абдрахманов, 1983]. Возможности системы расширены с помощью набора ОЛИСП-пакетов. В системе реализованы развитые возможности работы с данными, хранениями во внешней памяти. Помимо чтения и записи последовательных и библио-

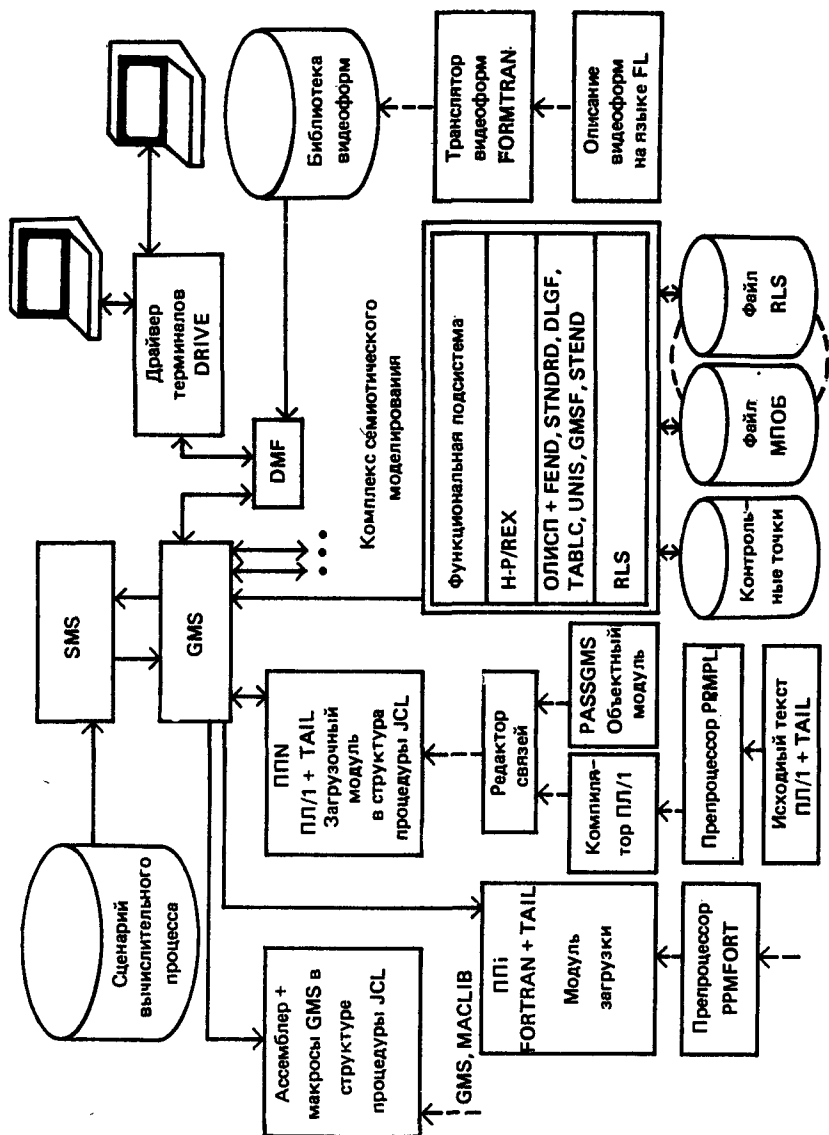
течных наборов данных, разработан и реализован метод доступа RLS, ориентированный на построение широкого спектра семиотических баз данных (БД) средствами языка ОЛИСП.

Уровень базового программного обеспечения (БПО) включает СПДЗ Н-Р/REX, представляющую собой систему расширенных продукций над сетевой моделью данных [Лозовский В., 1981]. С ее помощью реализуются базовые средства построения семиотических моделей предметных областей (МПОБ), обладающие достаточным диапазоном представления сложных информационных структур и гибкими средствами формализации операционного знания. Специальные средства ведения БД предусматривают ее организацию в виде фактуально-временного континуума, преобразования которого с помощью системы продукций — логико-трансформаторных правил (ЛТП) — управляются событиями, фиксируемыми в БД. Подобный способ запуска процедур, получивший название имediat-эффектного программирования, базируется на хорошо зарекомендовавшем себя механизме «демонов» [Лозовский В., 1984]. При реализации системы Н-Р/REX особое внимание уделено повышению ее быстродействия, что вызвано требованиями прикладного промышленного использования. Специальные средства: ассоциативность БД, жесткий механизм фокусировки при выборе релевантных ЛТП, снижающие до минимума переборы, а также тщательно сбалансированная трехуровневая система хранения для баз данных и знаний, включающая поля внутреннего ОЛИСП-представления, буферный пул в оперативной памяти и сетевую структуру, организованную средствами RLS, позволили для небольших по объему данных приложить при условии достаточного объема оперативной памяти практически всю работу выполнять без обращений к БД. Рост объема БД приводит к постепенному подключению эффективного механизма объектной виртуализации, учитывающего частоту обращений к объектам сетевой модели.

Опыт создания нескольких прикладных систем на базе рассматриваемых средств семиотического моделирования приводит к выводу о невозможности или во всяком случае о нецелесообразности построения единой СПДЗ на все случаи жизни. Целевая направленность пользовательских применений имеет слишком широкий диапазон, чтобы какие бы то ни было универсальные средства могли претендовать на удобство и эффективность применения. Разные задачи требуют создания своих парадигм представления, средств поддержки и функционирования. Этим и объясняется необходимость в следующем уровне ПО комплекса ИТК-БИС — функциональном. Функциональное программное обеспечение (ФПО) ориентируется на решение определенных целевых задач: управление вычислительным процессом, планирование, контроль, диагностирование, расчетные задачи и т. п. Помимо названных уровней существует еще один — модельное программное обеспечение (МПО), представляющее собой результат информационной специализации ФПО для решения определенного класса задач в условиях конкретных исходных данных и экспертной информации.

Общая структура комплекса ИТК-БИС представлена на рис. 3.6.

Функционирование комплекса включает две основные фазы: подготовительную и рабочую. Объем и содержание подготовительной фазы зависит от этапа, на котором начата эксплуатация комплекса. Если считать, что необходимая подсистема ФПО готова, то основная задача — построение МПОБ. Ее описание на исходном языке, определяемом соответствующей функциональной подсистемой, заносится в файл МПОБ, который может быть обособленным или погруженным в среду RLS. Одновременно проектируются на языке FL описания видеоформ, которые необходимы для работы модели, они обрабатываются транслятором видеоформ FORMTRAN и записываются в специальную библиотеку. Составляется набор вспомогательных модулей прикладного, расчетного, оптимизирующего характера и уточняется их зависимость друг от друга по данным и управлению. Устанавливается и уточняется описание сценария функционирования всего комплекса в целевой конфигурации. Он может быть очень простым для простых применений и сколь угодно сложным, учитывающим все необходимые условия совместного функционирования подготовленных подсистем. Этот этап, по сути, является составлением плана работы средств автоматизации функций оператора



вычислительного комплекса. Синтезированный сценарий записывается в систему H-P/REX, и на этом подготовительная фаза завершается.

В начале рабочей фазы использования комплекса ИТК-БИС запускается монитор GMS и считывается файл инициации, включающий в решение те задачи, которые должны функционировать безусловно. Затем иницируется монитор SMS. Дальнейшие действия всего комплекса разворачиваются в соответствии со сценарием (сценариями) и с учетом поступающих извне по каналам связи или от операторов данных и команд.

Модульно-стратифицированная структура комплекса ИТК-БИС позволяет использовать его по частям. Это оправдано и с методологической, и педагогической точки зрения. Пользователь при этом постепенно знакомится с возможностями тех или иных средств, обучается их использованию для своих целей, создает определенный задел программных наработок. Дальнейшее расширение возможностей при этом воспринимается как необходимое, мотивируется самим пользователем, а не «внедряется» извне. При этом переход на более полные конфигурации проходит естественно.

### **Инструментальная система программирования ОЛИСП**

При создании интеллектуальных автоматизированных систем, основанных на представлении знаний, возникает необходимость в более гибком языке программирования, чем традиционные языки. Одним из таких общепризнанных в области систем искусственного интеллекта инструментальных языков программирования является Лисп.

В качестве прототипа для создаваемой инструментальной системы был выбран ОЛИСП [Абдрахманов, 1983]. Основное отличие системы ОЛИСП от прототипа — расширение стандарта входного языка для режимов интерпретации и компиляции практически до уровня INTERLISP. При этом соблюдено почти полное вложение языка STANDARD LISP 1.5 в ОЛИСП (исключены для экономии памяти операции над числами неограниченной длины).

В состав системы программирования ОЛИСП входят: интерпретатор входного языка ОЛИСП; компилятор Лисп-функций и Лисп-ассемблер; средства диалоговой работы; средства взаимодействия с другими программными системами; препроцессор форматной печати текстов; расширяющие пакеты функций. С ориентацией на систему ОЛИСП в состав ОСПО включен специальный метод доступа RLS для реализации семиотических БД.

Для повышения быстродействия ОЛИСП при использовании в информационных системах большого объема значительно улучшены процедуры ввода и преобразования Лисп-выражений. Вместо последовательного сканирования OBLIST для преобразования односимвольных атомов применены машинные команды перекодировки по таблице и хешированный доступ к OBLIST в остальных случаях (эта возможность реализована О. Б. Воронцовой), что позволило повысить скорость преобразования выражений во внутреннее представление в 4—7 раз. Улучшены также средства вывода. В системе ОЛИСП применен более экономичный внутренних формат представления чисел и строк, что помимо экономии памяти позволило сократить время преобразования их во внутреннее представление. Введен новый тип данных — малое число (не более 3 байт). Для малых чисел не строится представляющая ячейка; вместо указателя на нее записывается само число. В ОЛИСП введена возможность определения программы выхода при исчерпании списочной памяти. Определенная пользователем Лисп-функция автоматически запускается в случае достижения задаваемого минимума числа свободных ячеек после срабатывания системного сборщика мусора. Это позволяет пользователю задавать свою программу «интеллектуальней» чистки памяти, основанную на предметном знании, либо, используя базу данных, организовывать семантическую виртуализацию. В систему ОЛИСП введены средства измерения времени, позволяющие измерять не только чистое процессорное время вычислений, но и время, затраченное на сборку мусора.

Компилятор, реализованный на языке ОЛИСП, позволяет получать быстродействующие Лисп-программы, практически не уступающие по эффективности

аналогичным программам на Ассемблере. Имеется также Лисп-ассемблер LAR, позволяющий включать в систему функции уровня Ассемблера.

**Средства диалоговой работы.** Система ОЛИСП может работать как в пакетном, так и в диалоговом режиме, используя дисплей ЕС-7920 (локальный или удаленный). Наличие интерпретатора и диалоговый характер языка позволяют в удобном режиме создавать, отлаживать и использовать разнообразные программы. Входной и выходной потоки системы ОЛИСП могут переключаться на дисплее независимо, что обеспечивает работу в комбинированном пакетно-диалоговом режиме. Имеются различные возможности управления выводом информации на экран. Информация может выводиться строками или кадрами в режиме автоматического роллинга (время экспозиции кадров задается пользователем) или роллинга с подтверждением. Можно прервать вывод на экран (например, в случае заикливания программы), нажав функциональную клавишу. При этом вычисление текущего выражения прекращается, и интерпретатор ОЛИСП переходит к выполнению следующего выражения на верхнем уровне. При вводе с дисплея система выдает комбинацию символов приглашения, которая может программно изменяться, отражая семантические уровни диалога. При работе в диалоге система автоматически ведет протокол диалога, который распечатывается в конце сеанса, протоколом можно управлять. В ОЛИСП реализованы средства реакции программы на сигнал «внимание», который может поступить от пользователя при нажатии им клавиши «Ввод» на дисплее. В случае сигнала «внимание» могут выполняться предусмотренные действия, например выход в режим диалога, создание контрольной точки.

**Работа с внешней памятью.** Система ОЛИСП имеет средства для обработки последовательных и библиотечных наборов ОС ЕС, а также средства создания Лисп-контрольных точек в виде последовательного файла, в котором сохраняется текущее состояние системы. С него может быть продолжено функционирование системы. Однако для организации эффективной обработки большого объема данных в системах семиотического моделирования этих средств оказывается недостаточно, что обусловило разработку и включение в состав системы ОЛИСП специального метода доступа RLS. RLS представляет собой простую и удобную в использовании, быстродействующую и универсальную систему хранения данных. RLS организует динамическое страничное распределение внешней памяти, позволяет записывать, считывать, модифицировать и удалять объекты произвольной длины как по имени (длиной до 256 байт), так и по относительному адресу на диске. Для ускорения доступа к объектам по имени используется хеширование и битовая карта страниц. Метод доступа RLS автоматически организует эффективную буферизацию и обеспечивает защиту данных при сбоях. Имеется возможность расширения файла RLS путем сцепления расширений (до 255 файлов), которые могут располагаться на дисковых устройствах различного типа (ЕС-5050, 5061, 5066). Для использования RLS в системе ОЛИСП имеется пакет соответствующих функций доступа, реализующих простой язык манипулирования данными. Метод доступа RLS используется как физический уровень хранения данных при реализации семиотических БД.

**Пакеты стандартных функций.** Система ОЛИСП может служить ядром расширенной системы программирования. Пример тому — различные пакеты функций, реализованных средствами ОЛИСП. В составе ОЛИСП имеются следующие пакеты функций: DLGF — для организации диалоговой работы; STNDRD — удобные стандартные функции обработки списков, печати, генерации уникальных атомов и др.; GMSF — интерфейс с мониторами GMS; RLSF — работа с методом доступа RLS; STEND — пакет функций сбора статистической информации о работе прикладных программ; TABL — пакет функций для составления таблиц.

Система ОЛИСП является мощной и эффективной комплексной реализацией Лиспа для ЕС ЭВМ, отвечающей требованиям промышленной эксплуатации. Система ОЛИСП сопровождается и развивается разработчиками, снабжена документацией. Ее применение позволяет упростить и ускорить этап программирования при создании базовых, функциональных и прикладных систем семиотического моделирования.



**Средства взаимодействия с другими программными системами.** Для организации межпрограммного интерфейса и построения сложных программных комплексов, состоящих из множества асинхронно выполняемых задач, в соответствии с заранее определенным сценарием ФПО может быть использован монитор GMS. Модули, работающие в архитектуре монитора GMS, могут быть написаны на Ассемблере, Фортране, ПЛ/1 и ОЛИСП. Для взаимодействия Лисп-программ с любыми задачами, функционирующими в архитектуре монитора GMS, в составе системы ОЛИСП имеется пакет интерфейсных функций, позволяющих посылать и принимать сообщения любой структуры. С помощью этих функций осуществляется взаимодействие с дисплейными модулями, использующими видеоформы, и любыми другими необходимыми программами.

### **Система представления данных и знаний Н-Р/REX**

Одна из главных проблем, возникающих при создании интеллектуальных систем, — это проблема представления знаний экспертов о предметной области в ЭВМ для использования их при принятии решений в сложных ситуациях.

Для описания и использования знаний экспертов традиционные языки недостаточны, так как отражают в основном вычислительную парадигму. Недостаточны также средства БД, так как в них семантика и динамика данных отделены от самих данных и вкладываются в прикладные программы на языках манипулирования данными (ЯМД). Недостаточность традиционных средств обусловила появление СПДЗ или баз знаний (БЗ), являющихся средствами построения интегральных семиотических МПОБ.

В составе комплекса ИТК-БИС таким базовым средством для построения семиотических моделей является СПДЗ Н-Р/REX, вобравшая опыт создания и использования семейства СПДЗ типа Н-Р [Лозовский В., 1979, 1981, 1984]. На базе СПДЗ Н-Р/REX строится МПОБ в виде продукционной системы над фактуальной БД в виде сетевой модели. Отличительная черта СПДЗ Н-Р/REX — гибкость, универсальность, эффективность, ориентация на реальные промышленные применения.

СПДЗ Н-Р/REX реализована в системе программирования ОЛИСП и включает средства ведения фактуальной БД, построения системы продукций, организации фасетной таксономической структуры и др. Основным компонентом Н-Р/REX являются средства ведения ассоциативной фактуальной БД, которые представляют собой специализированную семиотическую СУБД, реализованную на базе специального метода доступа RLS.

Второй компонент СПДЗ Н-Р/REX — средства построения системы продукций для представления процедурного знания. Продукционная система строится на основе ЛТП и ассоциативной управляющей структуры, определяющей, какое правило должно быть проверено следующим. Ассоциатор ЛТП строится на основании заданных в ЛТП условий активации (необходимые условия), которые в совокупности с проверяемыми условиями определяют возможность срабатывания ЛТП.

Средства построения таксономических структур используются для приписывания объектам модели свойств из некоторого определенного множества и задания на множестве свойств таксономической структуры, т. е. введения кроме элементарных свойств (фасетов) дополнительных составных свойств (таксонов), определяемых через объединение других таксонов или фасетов [Lozovsky V., 1982]. Таксоны могут сопоставляться с помощью логических операций И, ИЛИ, что позволяет строить классификационные решетки.

Кроме этих основных компонентов в системе Н-Р/REX имеются сервисные средства отображения, интерфейса с другими программными модулями, сбора статистики и трассировки.

**База данных Н-Р/REX.** Служит для хранения информационно-структурных знаний МПОБ, задания состояний модели и истории изменения состояний модели во времени, является единым средством обмена информацией между процессами, источником активации, синхронизации и реализации возникающих си-

гуаций. Концептуально БД организована в виде сетевой модели, в которую увязываются объекты и предикативные фреймы МПОБ. База данных может располагаться как в оперативной, так и во внешней памяти. Н-Р/РЕХ эффективно использует выделенную оперативную память для повышения производительности.

Логической единицей данных в Н-Р/РЕХ является атрибутивная структура — факт. Атрибутивной парой или атрибутом называется последовательность из двух элементов вида «имя атрибута — значение». Порядок записи атрибутов и фактов роли не играет. Наряду с атрибутивным форматом фактов используется позиционный, при котором имена атрибутов заменяются их номерами 1, 2, ...,  $N_i$  и в явном виде в фактах не задаются. При этом факты записываются в виде упорядоченной  $n$ -кн, где на  $j$ -й позиции находится значение  $j$ -го атрибута из  $DOM(A_j)$ .

База данных Н-Р/РЕХ представляет собой объединение экстенсиналов всех фреймов, введенных в МПОБ. Объединение схем фреймов БД представляет собой схему БД, которая является основой БЗ. Все факты БД получают уникальные метки, которые присваиваются системой и служат для ссылок на конкретный факт. Для факторов, хранимых во внешней памяти, метка факта играет роль адреса хранения.

Важнейшая особенность БД — ее ассоциативность и семантичность. Ассоциативность БД достигается с помощью автоматически организуемой системы инвертированных указателей — мультисети [Лозовский В., 1978]. С помощью мультисети реализуется эффективный поиск и выборка данных по образцу.

Запрос к БД оформляется в виде прототипа требуемой ситуации, представляющего собой образец искомого в БД множества фактов. Прототип задается в виде совокупности предложений, каждое из которых описывает требуемый факт БД. В предложении прототипа для каждого атрибута факта может указываться требуемое значение либо  $N$ -переменная, которая в случае успешного поиска конкретизируется, получая значение соответствующего атрибута из факта. Различаются декларация  $N$ -переменных и использование их значений. Декларация  $N$ -переменных может связываться с проверкой необходимых условий (например, вхождение значения в заданный интервал или наличие у него определенных свойств по таксономической структуре). С помощью механизма  $N$ -переменных предложения прототипа могут увязываться для описания любой необходимой ситуации в БД. Вторая функция  $N$ -переменных — извлечение необходимой информации из БД. При успешном поиске декларативные в прототипе  $N$ -переменные будут означены, т. е. получат значения соответствующих атрибутов из найденных фактов. Формат факта и предложения прототипа соотносятся следующим образом:

Факт:

((метка\_факта  $S_1 \dots S_k$ ) ИМЯ\_ТИПА  $A_1 \dots A_m \dots A_n$ )

Предложение		↑		↑		↑		↑
прототипа:								

( $N$ -имя\_факта СФЕРА ИМЯ\_ТИПА  $A_1 \dots A_m$ )

Здесь  $N$ -имя\_факта — это  $N$ -переменная, значением которой после успешного поиска будет метка факта, релевантного данному прототипу; СФЕРА, ИМЯ\_ТИПА,  $A_1, \dots, A_m$  в прототипе могут быть терминами (число, идентификатор, ЛИСИ-список), что требует наличия идентичного термина в соответствующей позиции факта, но, кроме того, они могут быть декларациями  $N$ -переменных либо  $N$ -значениями. Допускаются укороченные прототипы, т. е.  $M < N$ .  $N$ -декларации в предложениях прототипа могут иметь следующий вид:

(:) — пустая декларация, используется для позиционирования атрибутов, ей удовлетворяет любое значение атрибута факта

(:  $N$ -ПЕРЕМ) — простая  $N$ -декларация, при успешном поиске объявленная  $N$ -ПЕРЕМ получит значение соответствующего атрибута факта

(: Н-ПЕРЕМ УСЛОВИЕ) — условная Н-декларация, для успешного означивания требуется выполнение заданного УСЛОВИЯ, которое может быть произвольным предикатом на Лиспе, проверяющим значение Н-ПЕРЕМ и/или любых других переменных, получивших значение

Н-значения задаются в виде (Н-ПЕРЕМ). При поиске производится подстановка значения Н-ПЕРЕМ в качестве термина в прототип. Факты БД могут объединяться в произвольные подмножества (называемые сферами), которые могут пересекаться, т. е. каждый факт может одновременно входить в любое число сфер. Предусмотрены операции манипулирования такими множествами фактов.

**Язык манипулирования данными.** Для работы с БД имеется полный набор операций, составляющих ЯМД.

Ввод фактов в БД осуществляется с помощью функции ADE\*: (ADE\* (C ОТН A1 ... An)), где C — метка сферы (список меток сфер), в которую вводится факт; ОТН — имя типа фрейма; A1, ..., An — значения атрибутов факта.

Удаление фактов из БД осуществляется функцией DLF\*: (DLF\*L), где L — список меток удаляемых фактов или имя сферы, во втором случае удаляются все факты, входящие в данную сферу.

Работа со сферами производится с помощью функции TRS\*: (TRS\*M N L), где L — список меток фактов или метка сферы, задающая множество обрабатываемых фактов; M — метка сферы, из которой факты удаляются; N — метка сферы, в которую факты добавляются.

Модификация фактов выполняется функцией REPA: (REPA X L), где X — метка модифицируемого факта, а L — список замен вида (...N-атрибута новое значение...).

Для поиска и выборки фактов из БД используются функции IF и IFU, которые выполняют поиск заданных прототипом ситуаций, понимаемых как некоторое подмножество фактов БД:

(IF p1 ... pk)  
(IFU ACTION p1 ... pk),

где p1 — предложения прототипа; ACTION — некоторое действие, выполняемое при нахождении каждой конкретизации. Функции IF и IFU — предикаты, т. е. значение их в случае успешного поиска — T, в противном случае — NIL. Функция IF — аналог квантора существования, заканчивается с успехом в случае нахождения в БД первой же ситуации, удовлетворяющей заданному прототипу. Функция IFU — аналог квантора всеобщности, находит все ситуации, удовлетворяющие прототипу, и для каждой из них выполняет заданное при вызове действие ACTION в контексте значений Н-переменных каждой конкретизации.

Кроме перечисленных основных функций ЯМД включает их немедiated-эффективные аналоги и различные макромодификации, функции печати фактов, создания контрольных точек и др.

**Оперативная и внешняя БД.** База данных Н-Р/REX позволяет гибко учитывать требования конкретной предметной области при создании эффективных информационных систем. С этой целью в нее введены средства работы с оперативной и долговременной БД.

Множество фактов БД можно разделить по способу их использования на два класса:

оперативные факты, создаваемые и используемые только в течение одного сеанса и, как правило, требующие большой скорости доступа в связи с их частым использованием;

долговременные факты, существующие в течение нескольких сеансов работы БД.

В ходе сеанса работы с БД все факты и структуры мультисети, к которым производились обращения, загружаются с внешней памяти в оперативную и

остаются там, что увеличивает скорость доступа к ним при последующих обращениях. При исчерпании выделенной оперативной памяти автоматически запускается ситуационный мусорщик, который удаляет из оперативной памяти ненужные фрагменты БД, вытесняя при необходимости их во внешнюю память. Имеется возможность задания пользователем собственного алгоритма определения ненужных фактов в зависимости от модели предметной области и текущей ситуации. Такой объектно-ориентированный механизм виртуальной памяти данных, учитывающий семантику, приводит к увеличению эффективности по сравнению с обычными «неинтеллектуальными» механизмами виртуализации.

Физический уровень системы управления фактуальной БД построен в виде специального метода доступа RLS, представляющего собой реализацию объектно-ориентированной виртуальной памяти со страничной организацией для поддержки семиотических СУБД. Метод доступа RLS ориентирован на использование в составе СП ОЛИСП и реализован на языке Ассемблер. Для использования его имеется пакет функций, составляющих язык методов доступа RLS (ЯМД RLS), имеющий простые средства записи, чтения, удаления объектов как по имени, так и по адресу (R-коду), создания контрольных точек. RLS осуществляет функции динамического управления внешней памятью, создание и автоматическую работу с буферным пулом, создание контрольных точек и восстановление при сбоях.

**Модель данных.** СПДЗ Н-P/REX обеспечивает работу с сетевой моделью данных. Атрибутивная структура предикативных фреймов декларируется с помощью экземпляров фрейма RELAN, имеющих формат

((...метки\_сфер...) RELAN имя\_типа\_фрейма  
последовательность\_имен\_атрибутов)

Так, например, описание

((...KE...) RELAN МЕЖДУ что чем1 чем2)

декларирует структуру фрейма МЕЖДУ. Факты (экземпляры) вводятся в систему в атрибутивном или позиционном формате. В последнем случае обязательно соблюдение их порядка, предписываемого декларацией фрейма. Значения атрибутов в фактах могут быть идентификаторами, числами, строками и списками.

Структура абстракций в Н-P/REX задается на уровне отнесения фактов к определенным типам, при этом используется фасетная таксономия на именах типов.

**Представление времени.** Специальные средства представления времени позволяют строить фактуально-временной континуум состояний модели, работать с временными средами, учитывать историю протекания процессов, планировать будущие события и т. п. По способу привязки фактов БД к модельному времени в БД Н-P/REX различают нетемпоральные и темпоральные. Нетемпоральные факты — это факты, истинные в течение всего периода моделирования. Темпоральные факты делятся на факты-события и факты-состояния. Факты-события описывают мгновениые (с точки зрения целей моделирования) изменения модели в некоторый момент времени, в них зафиксирована семантика первого атрибута → он задает момент времени возникновения этого события. Факты-состояния используются для описания состояний сущностей модели предметной области и привязки их к интервалам существования. Обычно эти интервалы ограничены моментом событий начала и конца данного состояния. В фактах-состояниях зафиксирована семантика двух первых атрибутов: ТВ — время начала и ТЕ — время конца интервала существования.

Атрибуты времени в темпоральных фактах могут быть определены (заданы целыми числами) либо иметь неопределенное значение NIL, которое в дальнейшем может быть заменено определенным значением. Текущее модельное время моделируется с помощью системой переменной ГТС, значение которой автоматически меняется с ходом времени. Актуальные в текущий момент времени факты-состояния в качестве атрибута ТЕ могут иметь символ ГТС, что описывает их существование в текущем времени.

Пусть, например, в момент времени  $TB=186$  параметр  $P5$  перешел в состояние 1 и продолжает оставаться в этом состоянии. Это можно описать с помощью факта-состояния

(F5 B) P5 186 ГТС 1)

С течением модельного времени изменяется значение переменной ГТС, но факт F5 будет оставаться истинным. И лишь в момент перехода параметра  $P5$  в новое состояние этот факт будет «закрыт», т. е. значение атрибута TE будет изменено с ГТС на фактическое текущее значение ГТС. Таким образом, в ходе функционирования системы в БД накапливается история изменений состояний модели предметной области, которая может быть использована в процедурах принятия решений.

При работе с будущими событиями и состояниями в СПДЗ Н-Р/REX применяется механизм запланированных (отсроченных) событий (подробнее будет описан при рассмотрении имediat-эффектов).

**Управление модельным временем.** СПДЗ Н-Р/REX ориентирована на поддержание модели предметной области в виде фактуально-временного континуума состояний, который по отношению к текущему модельному времени ГТС делится на прошлое, настоящее и будущее. Для моделей, которые будут функционировать в реальном масштабе времени, важнейшей задачей является обеспечение синхронности функционирования модели и реальных объектов. Для такой синхронизации в СПДЗ Н-Р/REX введены средства автоматического продвижения текущего модельного времени в темпе поступления в модель событий из предметной области. Для этого необходима привязка вводимых в систему фактов ко времени снятия описываемых ими измерений, событий или ситуаций, т. е. представление их в формате событий. Если на вход системы поступает темпоральный факт  $F$ , то анализируется его время возникновения  $TB(F)$ . Если оно относится к прошлому ( $TB(F) < ГТС'$ ), то модельное время не изменяется. Если  $TB(F) = ГТС'$ , то  $F$  заносится в БД и активизируется соответствующий ему имediat-эффект определенного типа. Если факт относится к будущему ( $TB(F) > ГТС'$ ), то необходимо продвинуть текущее модельное время. Перед изменением ГТС система проверяет список запланированных на будущее событий и реализует те из них, время которых меньше (равно) полученного реального времени  $TB(F)$ . После этого значение ГТС становится равным значению  $TB(F)$ . Если  $F$  — синхронимпульс, то его обработка на этом заканчивается, иначе факт  $F$  заносится в БД и активируются соответствующие ему имediat-эффекты.

**Продукционная система.** Кроме информационно-структурных знаний об объектах, представляемых в фактуальной БД, в МПОБ обычно требуется фиксировать знания о процессах, причинно-следственных связях, законах функционирования, сценариях деятельности и т. п. Естественной формой представления таких знаний, наиболее соответствующей их процедурному характеру, является формализм ЛТП, представляющих собой расширенные продукции, Логико-трансформационные правила — это элементарный модуль процедурного знания, задающий действия по изменению (трансформации) состояния модели в случае возникновения определенной ситуации, гармонично дополняющий МПОБ сетевого типа средствами описания динамики и зависимостей, присущих природе предметной области.

Продукционная система на базе ЛТП, создаваемая средствами СПДЗ Н-Р/REX, включает три составляющих: фактуальную БД; множество ЛТП; управляющую структуру. Организация фактуальной БД Н-Р/REX была описана выше. В продукционной системе она является основным средством отображения состояния МПОБ, источником возникновения ситуаций, активирующих ЛТП и определяющих способ их срабатывания, а также единственным средством обмена информацией между процессами и их синхронизации.

ЛТП представляет собой элементарный квант процедурного знания, являясь, по сути, программой из одного оператора вида

ЕСЛИ <условие> ТО <действие> .

Каждое ЛТП состоит из прототипной и операционной частей. В *прототипной части ЛТП* задается условие в виде описания класса ситуаций применимости данного ЛТП по состоянию фактуальной БД. При возникновении соответствующей ситуации происходит срабатывание ЛТП. При написании прототипной части ЛТП задаются два типа условий: активирующие и проверяемые. Активирующие условия используются системой для построения управляющей структуры, с помощью которой определяется, какое ЛТП должно быть проверено следующим в сложившейся ситуации, для задания структуры моделируемых процессов, таких, как цепочки причинно-следственных связей, сценарии процессов контроля и управления и т. п. Проверяемые условия служат для описания необходимого для срабатывания данного правила контекста состояния БД такой степени детализации, которая требуется для верной передачи семантики данного процедурного кванта знаний.

*Операционная часть ЛТП* представляет собой описание алгоритма трансформации описания модели на инструментальном языке программирования ОЛИСП. Здесь, как правило, содержатся операции по добавлению, модификации или удалению фактов БД, а также выполняются необходимые вычисления, которые, возможно, используют другие программные модули, отображение информации на печать, дисплей и т. д.

Третий компонент продукционной системы СПДЗ Н-Р/РЕХ — *управляющая структура, или интерпретатор правил*, который отбирает из совокупности ЛТП «адекватные текущему состоянию модели и организует их активацию. Для эффективной организации управления правилами применяется ассоциатор ЛТП, который строится на основе прототипов активации, составляющих активирующие условия ЛТП. Коллизии по управлению между ЛТП разрешаются в порядке возрастания их приоритетов. Управляющая структура позволяет задавать стратегии, зависящие от контекста. Во многих задачах существуют различные режимы работы модели, каждая из которых характеризуется своим множеством ЛТП. Режим задается некоторой совокупностью условий — контекстом, в котором осмысленно применение некоторой группы ЛТП. Для фрагментации совокупности ЛТП на подмножества используется механизм режимов. При вводе ЛТП ему приписывается метка (метки) некоторого режима. Текущий режим модели задается значением системной переменной RGM. Для каждого режима строится отдельный ассоциатор ЛТП, а интерпретатором используется тот, который соответствует текущему режиму. В результате будут активироваться ЛТП, относящиеся только к текущему режиму.

**Иммедиа-эффектное программирование.** Продукционная система СПДЗ Н-Р/РЕХ, основанная на принципах гетерархии, ориентирована на технологию реализации интегрированных семиотических моделей, отражающих и статические, и динамические аспекты МПОБ. Эта технология заключается в проектировании и включении в модель иммедиа-эффектов, отражающих законы соответствующей предметной области. Под *иммедиа-эффектом* понимается эффект возникновения в модели последовательного процесса взаимообусловленных изменений, вызванный некоторым начальным изменением БД.

При поступлении в систему нового факта возможны три вида реакции СПДЗ. Во-первых, этот факт может быть отвергнут системой из-за его избыточности или противоречивости. Эти проверки могут выполняться соответствующими ЛТП, реализующими контроль целостности модели, которые играют роль процедур баз данных в традиционных СУБД. Во-вторых, факт помещается в БД, но никакой реакции на это не требуется, такие факты используются для полноты задания состояния модели. И, наконец, занесенный в БД факт требует немедленной реакции системы, обусловленной наличием в модели соответствующих возникшей ситуации причинно-следственных связей и законов, заданных в виде ЛТП. В этом случае занесение в БД фактов вызывает активацию и срабатывание соответствующих ЛТП, которые могут заносить в БД новые факты, а они, в свою очередь, могут возбудить срабатывание новых ЛТП и т. д. Такой процесс цепной реакции и изменение состояния БД и называется

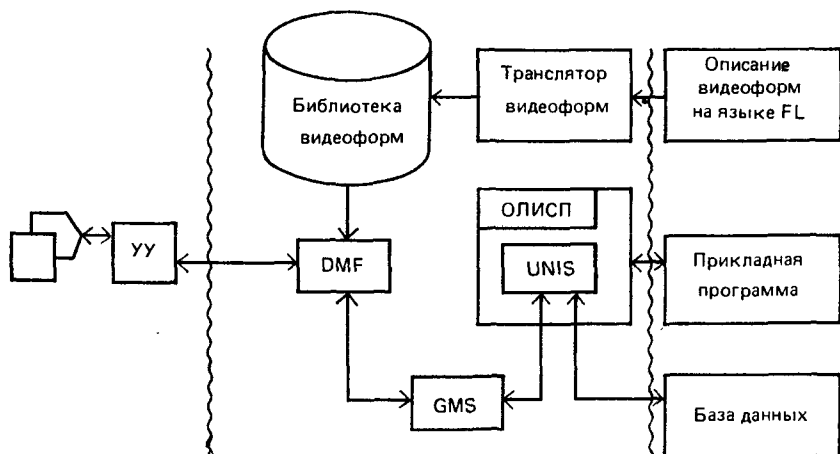


Рис. 3.7. Общая структура комплекса UNIS — GMS — DMF

ся имediat-эффектом. Иmediat-эффект обязательно должен быть конечным (затухающим).

Технология проектирования моделей в терминах имediat-эффекта, называемая имediat-эффективным программированием, обладает большой выразительной мощностью. Гибкость гетерархической системы в СПДЗ Н-Р/REX доведена до максимума благодаря применению анонимной активации модулей, полной локализации внутри ЛТП процедур проверки условий применимости и получения всех требуемых параметров, что делает ЛТП практически независимыми, а всю систему максимально модульной, легко расширяемой и модифицируемой.

С другой стороны, имediat-эффект можно рассматривать как реализацию логического вывода соответственно логике определенной предметной области. Доказательство некоторых «теорем» в такой логике может служить основанием для принятия верных оптимальных в определенном смысле решений по управлению, диагностированию и т. п.

Для диалогового доступа к БД СПДЗ Н-Р/REX предназначена программа UNIS (И. В. Ничолодини), позволяющая просматривать на экране содержимое БД, вводить с экрана в БД новые факты, удалять или корректировать уже имеющиеся. Программа UNIS работает совместно с дисплейным модулем DMF под управлением монитора GMS. Общая структура диалогового комплекса UNIS—GMS—DMF изображена на рис. 3.7.

Дисплейный модуль DMF осуществляет ввод-вывод информации через терминал и ее первичную обработку, монитор GMS обеспечивает совместное параллельное функционирование прикладной задачи программиста, модуля DMF и координирует обмен сообщениями между ними. Работа программиста с БД в диалоге с программой UNIS происходит через дисплей.

Программа UNIS написана на языке Лисп и в откомпилированном состоянии занимает около 5 Кбайт оперативной памяти. Время реакции программы UNIS в большой степени зависит от сложности запроса, объема БД и местоположения БД Н-Р/REX (оперативная или внешняя память).

Ввод-вывод информации на экране дисплея осуществляется посредством видеоформ. Видеоформы представляют собой бланки и таблицы фреймов, структура их стандартизована. Каждая видеоформа разделяется на две части: информационную и служебную. Информационная часть (она значительно больше

служебной) предназначена для ввода-вывода информации из БД; служебная часть видеоформы содержит необходимые пояснения относительно возможных действий на текущем шаге диалога, а также строку системных сообщений, в которой программа UNIS сообщает о допущенных ошибках, результатах выполнения определенных действий или причинах их невыполнения. Бланк фрейма предназначен, во-первых, для ввода новых фактов в БД и, во-вторых, для задания прототипа выборки фактов, подлежащих удалению из БД или просмотру, и, наконец, для выдачи по запросу единственного релевантного факта. При просмотре групповой выборки фактов в таблицах можно пользоваться навигационными операциями, управляющими движением по выборке.

### **Монитор GMS операционной интеграции задач в ОС ЕС**

Монитор GMS представляет собой единый загрузочный модуль GMSMAIN, который имеет монитор, обрабатывающий служебные сообщения, и набор реентерабельных подпрограмм (экстракодов), которые используются всеми задачами, работающими с GMS, для создания, отсылки, получения и обработки сообщений.

Монитор GMS функционирует на стандартном комплекте ЕС ЭВМ под управлением ОС MYT 6.1 и выше, в частности на SVS и CBM/BOC. GMS написан на Ассемблере и занимает 22 Кбайт оперативной памяти без учета объема рабочей области под очереди и сообщения, размеры которой задаются при запуске монитора.

В системе CMS использование монитора обеспечивают ряд элементарных операций, синхронизируя доступ к рабочей области и позволяя пользователю абстрагироваться от ее физической структуры: выделение памяти в рабочей области; освобождение этой памяти; запись сообщения (или его части); чтение сообщения (или его части); установление указателя для обеспечения возможности читать/писать сообщение с нужного байта; постановка сообщения в очередь к другой задаче (в том числе к монитору); вывод сообщений из своей входной очереди; поиск и, если требуется, ожидание появления нужного сообщения в своей входной очереди. Каждой такой операции соответствует свой экстракод, являющийся реентерабельной подпрограммой. Экстракоды обеспечивают защиту сообщений, каждое из которых в любой момент времени принадлежит только одной задаче.

Для задач GMS формирует входную очередь (одну для всех шагов данной задачи). Пересылка информации от задачи к задаче заключается в постановке сообщения, сформированного в первой задаче, во входную очередь. Так как одновременно могут выполняться несколько задач (экземпляров заданий) с одним именем, при запуске каждой задаче присваивается уникальный шифр, позволяющий однозначно указать адресата. Пересылаемые сообщения могут быть информационными и служебными. Первые используются для передачи информации между прикладными задачами, вторые — для связи с монитором. С помощью информационных сообщений могут передаваться только простые типы данных: целые числа, числа с плавающей запятой и строки. Каждое данное имеет свой тег, формируемый GMS, что позволяет проводить контроль типов данных при приеме сообщения. Служебные сообщения содержат требования установления или разрыва информационных связей. Выборка сообщений может осуществляться задачей только из своей входной очереди в двух режимах. В режиме WAIT при отсутствии требуемого сообщения задача переводится в состояние ожидания и продолжает работу при его появлении. В режиме NOWAIT задача в состоянии ожидания не переводится и о наличии нужного сообщения можно судить по коду возврата экстракода выборки. Выборка может быть селективной, при этом, например, могут выбираться сообщения только от определенной задачи или с указанным идентификатором.

Системы обработки информации, позволяющие решать достаточно сложные задачи, как правило, являются интерактивными. Поэтому в архитектуру монитора введена задача дисплейного модуля (DMF), которая обеспечивает



обмен информацией между задачами и человеком-оператором. С точки зрения монитора DMF является обычной задачей, которая может посылать и принимать сообщения, выполняет функции виртуального дисплея. При этом для каждого типа дисплея необходимо, естественно, написать свой дисплейный модуль.

Для разделения семантической обработки информации и операций форматного ввода-вывода DMF снабжен специальным языком форм. Форма является описанием формата экрана с указанием полей для ввода, вывода и некоторой другой информации. При этом вся семантическая обработка сосредоточена в задаче пользователя, а операции форматного ввода-вывода — в дисплейном модуле. Описание формата экрана производится на языке FL [Лозовский С., 1985]. Это описание обрабатывается транслятором форм и помещается в библиотеку, используемую DMF. FL связывает характеристики каждого поля ввода-вывода с его идентификатором. Таким образом, в сообщении, передаваемом DMF, указывается имя формы и пары идентификатор поля — значение. При выводе в поле, указанное идентификатором, помещается полученное значение, при вводе по сигналу «Внимание» формируется аналогичное сообщение, состоящее из модифицированных полей. Вследствие использования такой структуры сообщений при изменении формата выдачи на экран и сохранении имен полей корректируется только описание экрана на FL, но не обрабатывающая программа.

Включение программных модулей на ПЛ/1 в архитектуру GMS потребовало включения специальных информационных структур и операторов. В процессе разработки интерфейса было принято решение в максимальной степени ограждать автора прикладных программ от нижнего уровня взаимодействия с монитором. С этой целью был разработан специальный язык межадачного взаимодействия TAIL-2, имеющий простую операторную структуру и являющийся расширением ПЛ/1. Выражения на TAIL-2 используются программистом в программе на ПЛ/1, после чего исходная программа обрабатывается препроцессором PPMPL, транслирующим ее в текст на ПЛ/1. Операторы TAIL-2 преобразуются в обращения к специальному интерфейсному модулю PASSGMS, подключаемому на этапе редактирования связей.

### Подсистема SMS

Подсистема SMS предназначена для организации вычислительного процесса в больших программных комплексах, состоящих из параллельно выполняемых задач. Вычислительный процесс может иметь очень сложную динамическую структуру, состояние и изменения которой определяются множеством факторов: это и текущая конфигурация вычислительных средств, их исправность, степень загруженности, состояние хода решения целевых, системных и сервисных задач, готовности тех или иных данных для использования соответствующими задачами, поступление исходных данных или запросов по внешним каналам, привязка к астрономическому времени по периодичности запуска тех или иных задач, учет необходимых временных интервалов и т. п.

Архитектура междоиспользования взаимодействия на базе монитора GMS операционной интеграции в принципе позволяет решить все эти задачи. В достаточно сложных системах вопросы общесистемного управления на уровне готовых функциональных модулей обычно не решаются, а часто и не могут быть решены из-за невозможности оценить вычислительную обстановку в целом. Поэтому в основу разработки подсистемы SMS были положены следующие принципы: максимальная «бережность» к уже готовым программным модулям; сведение описаний всех процессов и всех характеристических параметров в единую оперативную БД и обеспечение адекватного действительности отслеживания ее состояния; унификация знаний о ходе вычислительного процесса, правилах принятия решений и концентрация их в единой БЗ в такой форме, которая, с одной стороны, облегчила бы и всячески стимулировала процесс извлечения знаний из экспертов для своего создания и оперативной модификации, а с другой — обеспечивала бы эффективное внутрисистемное взаимодействие с БД

системы с целью своевременного принятия нужных решений по управлению; использование в качестве основы сценарной модели представления знаний.

Средством поддержки рассмотренных выше принципов проектирования SMS является язык описания вычислительного процесса SML, который позволяет описывать последовательность запуска задач и схемы коммутации информационных связей между ними. Описание вычислительного процесса на языке SML называется сценарием. В процессе работы SMS интерпретирует сценарии вычислительного процесса, используя в качестве исходных данных SML-описания и информацию, поступающую от ОС и выполняемых задач. На выходе SMS—управляющие воздействия на ОС (запуск задач), сообщения, посылаемые задачам в соответствии со сценарием, и протокол выполнения сценария. SMS позволяет анализировать сообщения, поступающие от задач, и направлять задачам сообщения. Поскольку среди задач может быть дисплейный модуль (и), то с помощью SMS может быть описан и диалог с оператором. SMS функционирует под управлением ОС MVT, SYS 6.1/2 и выше, БОС CBM и позволяет использовать как задачи, так и подзадачи.

Описание вычислительного процесса удобно представлять в виде графа, каждая вершина которого может иметь несколько точек входа и выхода. В процессе работы SMS на выходах вершины может возникать информация, которая передается на соответствующие входы других вершин по исходящим дугам. Весь граф называется сценарием вычислительного процесса, а вершины—подсценариями. Передача информации по дуге называется передачей активности. Таким образом, активация некоторых выходов одного сценария влечет за собой активацию некоторых входов другого. В свою очередь, активация некоторых входов сценария при определенных условиях может вызвать активацию его выходов. Информация на входы сценария может поступать асинхронно, информация на выходах также может появляться не одновременно. В SMS поддерживаются сценарии трех типов:

сценарий типа «условие» — явно записывается логическое выражение зависимости выходов от входов; если логическое выражение становится истинным, то на все выходы этого сценария передается активность, а активность с его входов снимается; проверка условия производится при активации каждого из входов;

терминальный или встроенный сценарий — не имеют графового представления и являются, по существу, встроенными процедурами SMS;

составной сценарий — является графом, вершинами которого могут быть условия, терминальные и составные сценарии, а передаваемая активность может быть двух типов: активность-событие (или просто активность) — при передаче на вход условия снимается с этого входа в случае выполнения условия, и активность-состояние (или просто состояние) — не снимается при выполнении условия. Снятие активности состояния производится путем активации соответствующего входа значением NIL.

Язык описания сценариев SML. Термины SML записываются латинскими буквами, а нетерминальные понятия — русскими. Двоеточие предвещает определение стоящего перед ним нетерминального понятия, а троеточие — возможность повторения стоящей перед ним конструкции. Вертикальная черта означает обязательный альтернативный выбор. В квадратных скобках указываются конструкции, которые могут отсутствовать, угловые скобки ограничивают область действия вертикальных черт и троеточия. Вертикальная черта, условие и квадратные скобки, двоеточие, троеточие и запятая являются метасимволами.

Описание вычислительного процесса всегда состоит хотя бы из одного составного сценария:

ОПИСАНИЕ\_ТИПА\_СЦЕНАРИЯ:

<ОПИСАНИЕ\_ТИПА\_СОСТАВНОГО\_СЦЕНАРИЯ|  
ОПИСАНИЕ\_ТИПА\_ТЕРМИНАЛЬНОГО\_СЦЕНАРИЯ>

ОПИСАНИЕ\_ТИПА\_СОСТАВНОГО\_СЦЕНАРИЯ:

(ТИПЕ ИМЯ\_ТИПА<ОПИСАНИЕ\_УСЛОВИЯ|ОПИСАНИЕ\_ТИПА\_

СЦЕНАРИЯ>...)  
ОПИСАНИЕ\_УСЛОВИЯ:  
(МЕТКА IF ПРЕДИКАТ THEN АКТИВАЦИЯ\_ТОЧКИ...)  
ПРЕДИКАТ:  
<ТОЧКА\_АКТИВАЦИИ|  
(ЛОГИЧЕСКАЯ\_СВЯЗКА ПРЕДИКАТ...)|СПЕЦФУНКЦИЯ>  
ТОЧКА\_АКТИВАЦИИ:  
(МЕТКА ИМЯ\_ВХОДА\_ВЫХОДА)  
ИМЯ\_ВХОДА\_ВЫХОДА, ИМЯ\_ТИПА, МЕТКА:  
ИДЕНТИФИКАТОР  
ЛОГИЧЕСКАЯ\_СВЯЗКА:  
<AND|OR|NOT>

NOT имеет один аргумент, а AND и OR — произвольное число. К специальным относятся следующие функции: EQUAL (равно), LESSP (меньше), GREATERP (больше), вырабатывающие значение T («ИСТИНА») или («ЛОЖЬ»):

СПЕЦФУНКЦИЯ:  
(<EQUAL|LESSP|GREATERP> АРГУМЕНТ1 АРГУМЕНТ2)  
АРГУМЕНТ1, АРГУМЕНТ2:  
<ТОЧКА\_АКТИВАЦИИ|СТРОКА|ЧИСЛО|  
(NUMA МЕТКА\_ТОЧКИ\_АКТИВАЦИИ1  
ИМЯ\_ВХОДА\_ВЫХОДА\_ТОЧКИ\_АКТИВАЦИИ1  
МЕТКА\_ТОЧКИ\_АКТИВАЦИИ2  
ИМЯ\_ВХОДА\_ВЫХОДА\_ТОЧКИ\_АКТИВАЦИИ2)>  
АКТИВАЦИЯ\_ТОЧКИ:  
(МЕТКА ИМЯ\_ВХОДА\_ВЫХОДА ЗНАЧЕНИЕ)

Функция NUMA вырабатывает число активации ТОЧКИ 1 после последней активации ТОЧКИ 2.

Например, пусть необходимо запустить задачу с меткой В после того, как задача с меткой А выполнялась три раза с начала работы сценария и сценарий был активирован с точки (ВХОД ЗАП). Это условие должно быть записано следующим образом:

```
(PI IF (AND (A КОН)
              (EQUAL (NUMA A КОН ВХОД ЗАП) 3))
  THEN ((В ЗАПУСТИТЬ Т)))
```

(А КОН) задает момент, когда должна быть выполнена проверка.  
ОПИСАНИЕ\_ТИПА\_ТЕРМИНАЛЬНОГО СЦЕНАРИЯ:  
(МЕТКА IS ИМЯ\_ТИПА [WITH СПИСОК\_ЗНАЧЕНИЙ\_НА\_ВХОДАХ]  
[ENTRIES СПИСОК\_ДОПОЛНИТЕЛЬ-  
НЫХ\_ВХОДОВ])  
СПИСОК\_ЗНАЧЕНИЙ\_НА\_ВХОДАХ:  
((ИМЯ\_ВХОДА <ЗНАЧЕНИЕ, ТОЧКА—АКТИВАЦИЯ>)...)  
СПИСОК\_ДОПОЛНИТЕЛЬНЫХ\_ВХОДОВ:  
(ИМЯ\_ВХОДА...)

Запуск интерпретации всего описания вычислительного процесса осуществляется с помощью оператора MAIN, имеющего следующий синтаксис:

```
(MAIN (АКТИВАЦИЯ_ТОЧКИ...) ОПИСАНИЕ_СЦЕНАРИЯ)
АКТИВАЦИЯ_ТОЧКИ:
(МЕТКА_ИМЯ_ВХОДА_ВЫХОДА ЗНАЧЕНИЕ [C])
МЕТКА, ИМЯ_ВХОДА_ВЫХОДА:
ИДЕНТИФИКАТОР
ЗНАЧЕНИЕ:
<ЧИСЛО|СТРОКА|ИДЕНТИФИКАТОР>
```

Если после значения стоит признак «С», то считается, что передается состояние:

ОБЪЯВЛЕНИЕ\_СЦЕНАРИЯ (СОСТАВНОГО):  
(МЕТКА IS ИМЯ\_ТИПА)  
ИМЯ\_ТИПА:  
ИДЕНТИФИКАТОР

Например, пусть в описании УПР\_ПР\_УЧ (управление производственным участком) задана точка входа НАЧ. Для запуска сценария требуется следующий оператор:

```
(MAIN ((A НАЧ Т))  
      (A IS УПР_ПР_УЧ))
```

Таким образом, интерпретация сценария начнется с активации точки входа НАЧ сценария А. Рассмотрим описание типа УПР\_ПР\_УЧ и запуск сценария А этого типа оператором MAIN:

```
(MAIN ((A НАЧ Т))  
      (A IS УПР_ПР_УЧ))  
(TYPE УПР_ПР_УЧ  
  (P1 IF (ВХОД НАЧ)  
    THEN ((БАЗА_ДАННЫХ ЗАПУСТИТЬ Т)  
          (УПР ЗАПУСТИТЬ Т))))  
  (БАЗА_ДАННЫХ IS ЗАДАЧА WITH ((ПРОЦ «BD»)))  
  (УПР IS ЗАДАЧА WITH ((ПРОЦ «CONTR»)))  
  (P2 IF (AND (БАЗА_ДАННЫХ КОН)  
          (УПР КОН))  
    THEN ((КОНЕЦ СЦЕНАРИЯ Т)))  
  (КОНЕЦ IS КОНЕЦ))
```

Сценарий КОНЕЦ имеет терминальный тип 'КОНЕЦ'. При передаче активности на вход «СЦЕНАРИЯ» (Условие P2) происходит завершение работы SMS. Составной сценарий типа УПР\_ПР\_УЧ имеет точку входа НАЧ, если этот сценарий используется в качестве подсценария, то в охватывающем его сценарии может быть активирован вход НАЧ (как это сделано в операторе MAIN). Чтобы получить активность со своего входа, при описании составных сценариев используется метка ВХОД (см. P1 в приведенном выше примере). Если составной сценарий имеет выходы, то используется метка ВЫХОД.

Терминальные сценарии описания вычислительного процесса. Терминальные сценарии составляют системную библиотеку SMS. Ниже рассматриваются основные терминальные сценарии.

1. Терминальный сценарий ЗАДАЧА. Основные входы этого сценария:

ПРОЦ — установить имя процедуры;

ПАР — установить параметры запуска;

ЗАПУСТИТЬ — при передаче активности на этот вход происходит запуск процедуры с именем ПРОЦ и параметрами ПАР; если вход ПАР не активирован, процедура запускается без параметров;

ОСТ — при активации этого входа задаче пересылается STOP-сообщение, после получения которого она должна окончить работу;

АДРЕСАТ — этот вход используется при установлении информационной связи;

ОТПРАВИТЬ — послать информационное сообщение задаче, представленной этим сценарием;

ОТН — имя отношения (видеоформы для DMF).

Дополнительные точки входа задают передаваемую информацию. Структура сообщений, передаваемых задаче, соответствует структуре сообщения, предназначенного для DMF. Выходы сценария активируются в следующих случаях:

ЗАП — успешный запуск задачи;

НЕ\_ЗАП — задача не запустилась (например, из-за JCL ERROR и других ошибок);

КОН — задача закончилась (нормально или аварийно); если задача не запустилась, этот вход не активируется;

ABS\* — задача завершилась аварийно;

ABS' — системный код ABEND;

ABU\*—ABU' — пользовательский код ABEND;

ВЫПОЛНЯЕТСЯ — активируется состоянием на время выполнения задачи;

СВЯЗЬ — задача прислала START-сообщение и просит установить связь; для установления связи активность с этого выхода должна быть передана на вход АДРЕСАТ другой задачи. Если задача-адресат не выполняется, то активизация входа АДРЕСАТ вызовет послылку отказа задаче, пославшей START-сообщение (в ответе будет установлен ненулевой код возврата);

ИМЯ\_С — имя процедуры (или данных), указанное в START-сообщении; ПОТН — имя обращения в предыдущем от задачи сообщении (формат принимаемого сообщения аналогичен посылаемому); дополнительные точки выхода активируются информацией из полученного сообщения.

Например, при окончании задачи А нужно послать сообщение <0 кончился модуль «А»> задаче В, а при получении от нее сообщения <0 ЗАП ЗАДАЧУ «С»> запустить задачу С. Для выполнения этих действий можно использовать следующий сценарий:

(В IS ЗАДАЧА ENTRIES (МОДУЛЬ))

(P1 IF (А КОН) THEN (В ОТН КОНЧИЛСЯ) (В МОДУЛЬ «А»)  
(В ОТПРАВИТЬ Т)).

(P2 IF (AND (EQUAL (В ПОТН) ЗАП)  
(EQUAL (В ЗАДАЧУ) «С»))  
THEN (С ЗАПУСТИТЬ Т))

2. Терминальный сценарий ПОДЗАДАЧА предназначен для запуска подзадач, а его структура аналогична структуре терминального сценария ЗАДАЧА.

3. Терминальный сценарий ИНФ предназначен для чтения информации из таблиц, находящихся в БД.

4. Терминальный сценарий типа СВЯЗЬ используется в тех случаях, когда необходимо иметь подробную информацию о связи между двумя задачами, т. е. каждой информационной связи может соответствовать свой сценарий. Терминальный сценарий типа СВЯЗЬ П предназначен для связи двух пассивных задач, т. е. таких, которые не выдают сами запросов на установление связи, а только отвечают на поступающие сообщения-запросы (например, БД и дисплейный модуль).

5. Терминальный сценарий типа КОНЕЦ активируется в конце работы сценария верхнего уровня для завершения работы SMS. Интерпретация сценария при этом завершается.

Программа SMS выполняется в архитектуре GMS как одна из подзадач. Ее отличие от других задач состоит в том, что все служебные сообщения, передаваемые задачами, попадают к SMS, а не к монитору GMS и обрабатываются в соответствии со сценарием. Связь с ОС (запуск задач, получение информации об их окончании), создание и уничтожение очередей и другие действия SMS осуществляет, обмениваясь сообщениями с GMS. Информация, поступающая SMS в виде сообщений от GMS и прикладных задач, после предварительной обработки заносится в БД (СПДЗ Н-Р/REX). В SMS имеются функции, осуществляющие передачу сообщений из SMS прикладным задачам и GMS. Монитор GMS обеспечивает использование в архитектуре задач ОС ЕС, а комплекс SMS/GMS — и подзадач. SMS позволяет всю специфическую информацию о программной системе собрать в сценарии, именно поэтому появляется возможность использования подзадач и построения системы на основе произвольного графа информационных связей.

### 3.4. Средства поддержки проектирования прикладных экспертных систем в оболочке СПЭИС

*О. В. Ковригин*

При создании прикладных экспертных систем (ЭС) с помощью СПЭИС для разработки базы знаний и интерфейса ЭС с пользователем разработчик может использовать формализм представления и интерфейс, уже имеющийся в системе-оболочке СПЭИС. Если предлагаемый формализм по каким-либо причинам не вполне устраивает проектировщика, то он может, используя примитивы формализма СПЭИС, создать свою версию языка представления знаний (ЯПЗ). При желании может быть изменен и интерфейс СПЭИС.

Средства поддержки разработки прикладных ЭС в СПЭИС можно разбить на две группы. Первая служит для создания и отладки непосредственно прикладных баз знаний, а вторая — для разработки новых модификаций ЯПЗ и создания нового интерфейса.

#### **Средства поддержки проектирования баз знаний**

Средства, входящие в эту группу, включают: пакет подготовки информации; пакет представления информации; пакет проверки структур баз знаний; пакет тестовых прогонов системы; пакеты прерываний и объяснений работы ЭС.

**Пакет подготовки информации.** Используется для создания информационного наполнения оболочки. Информация о проблемной области в СПЭИС представляется Н- или Р-концепциями или произвольными объектами. Создание описаний концепций и объектов можно производить двумя способами: используя встроенный редактор фреймов системы СПЭИС или любой текстовой редактор (например, Лексикон Веселова). Определенные концепции хранятся в файлах последовательного доступа. Базу знаний можно хранить в одном файле или разнести группы концепций по различным файлам.

Неподготовленный пользователь, используя экранный редактор фреймов, получает некоторые преимущества. В ЯПЗ часто для выделения области действия логических операций используются скобки. Редактор следит за балансом скобок и всегда выделяет на экране незакрытую скобку другим цветом символов. Можно использовать специальный редактор правил при определении в теле концепции правил. Кроме того, работая с встроенным редактором, пользователь может просматривать описание других концепций, анализировать их отношения, что упрощает весь процесс создания базы знаний.

**Пакет представления информации.** Служит для отображения структур БЗ в формах, наиболее приемлемых для разработчика и/или эксперта — текстовой и графической. Каждая концепция неизбежно содержит сокращения и кодировки (например, имен концепций). В СПЭИС имеется специальное средство трансляции тела концепции в форму, близкую к естественному языку (ЕЯ). При этом на места закодированных элементов подставляются их ЕЯ-версии, скобки исчезают, а область действия логических операций определяется соответствующим сдвигом текста. Режим изображения концепций можно переключать по желанию пользователя. Пример одного из таких изображений:

(DS P4

(IS-A. P-КОНЦЕПЦИЯ)

(PHANE САМЫЕ ВЫСОКИЕ ЦИФРЫ ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ)

(POSSIBLE-VALUE. number)

(RESTRICTION (between 0 280))

(DESCRIPTION P6 P16)

(FRESTRICION (less-then P3))

(STR-CONTROL

```
(1 ((IF (AND (P3<=P5) (P4<=P6))) (THEN (ACTIVATE: END)
(STOP:))))]
```

Концепция Р4

имя

**САМЫЕ ВЫСОКИЕ ЦИФРЫ ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ**

возможные значения ЧИСЛО ограничения МЕЖДУ 0 280

ограничения! МЕНЬШЕ ЧЕМ САМЫЕ ВЫСОКИЕ ЦИФРЫ СИСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ

ассоциативная связь

**ВОЗРАСТНАЯ НОРМА ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ**

**ЦИФРЫ ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ РАНЬШЕ**

управление выводом

1 ЕСЛИ И

**САМЫЕ ВЫСОКИЕ ЦИФРЫ СИСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ<=**

**ВОЗРАСТНАЯ НОРМА СИСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ**

**САМЫЕ ВЫСОКИЕ ЦИФРЫ ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ<=**

**ВОЗРАСТНАЯ НОРМА ДИАСТОЛИЧЕСКОГО АРТЕРИАЛЬНОГО ДАВЛЕНИЯ**

**ТО АКТИВИЗИРОВАТЬ Н-КОНЦЕПЦИЮ: ДАВЛЕНИЕ В ПРЕДЕЛАХ НОРМЫ  
СТОП:**

Текстовое изображение концепций позволяет эксперту и разработчику легко анализировать информацию внутри одной концепции, однако не позволяет проследить взаимосвязи между концепциями. Р- и Н-концепции могут образовывать связанные фрагменты сетей — кластеры. Н-концепции вызывают друг друга в Р-концепции, образуя деревья вывода решений. В СПЭИС разработан пакет графической визуализации связей между объектами. На экран дисплея (желательно цветного) отображается фрагмент сети относительно указанной корневой вершины. Размер сети не ограничен полем одного экрана. При этом задается тип связи между вершинами. Разработчик может, используя клавиатуру или устройство типа «мышь», перемещать курсор по вершинам сети.

Когда курсор находится в некоторой вершине сети, возможен просмотр описания концепции, соответствующей вершины. С помощью механизма наложения окон пользователь одновременно видит участок сети в одном окне экрана и описание вершины в другом окне. Построенные фрагменты сетей могут быть выведены и на печать. Два примера изображения сетей на экране приведены на рис. 3.8.

**Пакет проверки структур баз знаний.** Предназначен для синтаксической и семантической проверки баз знаний. Проверке может подвергаться как отдельная концепция, так и совокупность концепций ЭС. Проверка каждой отдельной структуры производится с целью определения: неопределенных Р- и Н-концепций; синтаксических ошибок в примитивах ЯПЗ; неопределенных функций; потенциально невыполнимых правил.

При анализе всех концепций баз знаний выделяются структуры, которые потенциально не могут быть активизированы вследствие допущенных синтаксических или логических ошибок. Приведем пример одного диагностического сообщения СПЭИС:

Правило 4 в концепции Феохромоцитома (FEONHR)

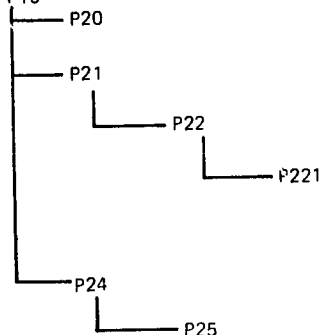
<4 если И

**ХАРАКТЕР ПОВЫШЕНИЯ АРТЕРИАЛЬНОГО ДАВЛЕНИЯ**

**ПАРАКСИЗМАЛЬНЫЙ**

**ВОЗРАСТ<40**

Кластер Р-концепций для корневой вершины P19 по ассоциативной связи:  
P19



Концепция P20

имя ВОЗРАСТ

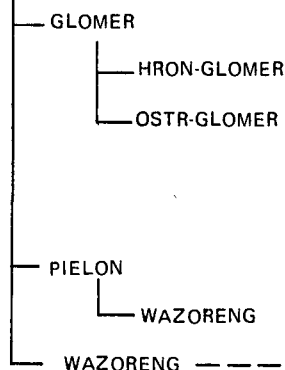
возможные значения ЧИСЛО

ограничения МЕЖДУ 0 100

а)

Дерево возможных вызовов для Н-концепций для корневой вершины  
НЕФРОЛОГИЧЕСКАЯ ГРУППА ЗАБОЛЕВАНИЙ

NEPHROLOGY



Концепция ГЛОМЕРУЛОНЕФРИТ

условие активизации

или КОЛИЧЕСТВО ЛЕЙКОЦИ  
КОЛИЧЕСТВО ЭРИТРОЦ

Концепция ВАЗОРЕНАЛЬНАЯ ГИП

условие активизации

и НАЙДЕН ПИЕЛОНЕФРИТ  
НЕ НАЙДЕН ГЛОМЕРУЛОНЕФ

б)

Рис. 3.8. Экранное представление Р-концепций (а) и Н-концепций (б) в системе СПЭИС

то ЗАКЛЮЧЕНИЕ: У больного не исключена феохромоцитома». невыполнимо, т. к. в область возможных значений параметра ХАРАКТЕР ПОВЫШЕНИЯ АРТЕРИАЛЬНОГО ДАВЛЕНИЯ (P7) не входит значение ПАРАКСИЗМАЛЬНЫЙ. ВОЗМОЖНЫЕ значения для P7 определены как список: (СТАБИЛЬНЫЙ ЛАБИЛЬНЫЙ КРИЗОВЫЙ).

**Пакет тестовых прогонов.** Позволяет проводить тестовые прогоны макета ЭС с целью определения недостатков базы знаний и их устранения. В СПЭИС реализована возможность тестовых прогонов в условиях частичного наполнения базы знаний, т. е. при наличии в базе знаний ссылок на неопределенные концепции и объекты. Парадигма вывода решений в СПЭИС обеспечивает ав-



томатическую блокировку неопределенных ссылок. Удобно отлаживать базу знаний как часть независимых сегментов знаний. Н-концепции, управляющие выводом решения, образуют связанные кластеры. Используя относительную независимость концепций, можно инициализировать вывод решения с любой вершины кластера. В этом случае вывод затронет только те концепции, которые могут быть активизированы из исследуемой, и не затронет концепций, относящихся к другим кластерам. Кластеры легко можно проанализировать, используя пакеты визуализации баз знаний.

**Пакеты прерываний и объяснений работы ЭС.** Позволяют разработчику в режиме консультации осуществлять постоянный контроль за состоянием системы в процессе вывода. Работа программы вывода может быть прервана в любой момент. При этом возможен доступ к следующей информации: текущие активные концепции и объекты; трасса вывода решений; состоящие части памяти, доступной всем концепциям; рабочая информация по состоянию концепций и объектов.

Список текущих активных концепций и объектов позволяет разработчику оценить состояние системы на момент прерывания и проанализировать, почему те или иные концепции активны.

Трасса вывода решений представляет собой последовательность примененных правил с указанием концепций, в которых они были выполнены. Анализируя эту последовательность, разработчик базы знаний совместно с экспертом может оценить, какие правила привели к желаемому результату, или определить, какое правило привело к ошибке в рассуждениях системы.

В процессе вывода решений формируются промежуточные решения, которые могут заноситься в общедоступную память, замещать другие или удаляться из памяти. Кроме того, в памяти находятся утверждения, сделанные программой вывода либо сформированные непосредственно на основании запросов системы и ответов пользователя. Вся совокупность информации в этой памяти доступна разработчику в любой момент.

При активизации любого объекта в процессе вывода решений накапливается разного рода промежуточная и служебная информация, необходимая программе вывода для оценки концепций. Это временные стеки решений и концепций-кандидатов на активизацию. Стеки очищаются сразу после окончания обработки концепции, но в них содержится интересная информация, полученная в процессе ее обработки. Служебная информация позволяет определить, какие концепции активизировали рассматриваемую, какие были активизированы из нее, какие правила «провалились» и почему, закрыта или открыта концепция для новой активизации и т. п.

Используя средства системы СПЭИС по поддержке процесса разработки прикладных баз знаний, можно в значительной мере сократить время проектирования баз знаний и количество неизбежных ошибок в знаниях ЭС.

### **Создание интерфейса экспертных систем**

Для современных ЭС характерно наличие дружелюбных интерфейсов. Система должна быть удобной для конечного пользователя, который, как правило, не имеет навыка общения с ЭВМ. Для реализации интерфейса разрабатываемой ЭС с пользователем в СПЭИС предлагаются следующие возможности: разработка контекстно-зависимой помощи; организация запросов к пользователю в режиме «меню»; многооконные режимы работы на дисплее; ввод и вывод информации в виде графических образов.

Контекстно-зависимая помощь необходима для пользователя, когда он находится в затруднительном положении при работе с программой. В СПЭИС каждому текущему состоянию системы должно соответствовать некоторое активное окно на экране дисплея (некоторый активный объект), предусмотрен специальный механизм связывания текущего состояния системы с множеством возможных действий в этом состоянии. Разработчик, определяя активные окна, может задать список функциональных клавиш, которые доступны пользовате-

лю при работе в этом окне. Кроме того, он определяет подпрограммы (или объекты), которые должны быть активизированы при нажатии соответствующих клавиш.

Самым простым способом общения пользователя с ЭС является режим диалога, в котором ЭС задает вопросы и предлагает пользователю выбрать один или несколько ответов. Для реализации такого диалога в СПЭИС разработан инструментарий задания меню с указанием размера и цвета окон, числом колонок расположения информации, множественным и единичным выбором ответов. Кроме того, реализованы достаточно эффективные средства рисования окон, их наложения, хранения, стирания и восстановления. Каждое окно определяется как некоторый объект со своими атрибутами. Такой объект может находиться в активном или пассивном состоянии, запоминать свое состояние до необходимого момента. Используя такой объектно-ориентированный стиль работы с окнами, достаточно легко спроектировать свой интерфейс взаимодействия с пользователем.

В некоторых приложениях ЭС наиболее адекватным способом общения с экспертом является общение с помощью графических изображений (картинок). В СПЭИС предоставляется возможность определять объекты, которые связываются с графической информацией. Каждый такой объект — это одно изображение, которое может быть использовано, например, как одно из возможных значений в Р-концепции. Работа с графическим изображением осуществляется путем вызова специальных подпрограмм, написанных на языке Си и использующих графические примитивы системы HALLO. Подготовить исходные изображения можно с помощью любого графического редактора (например, редактора OLPEN, реализованного автором).

Таким образом, система СПЭИС является достаточно развитой оболочкой для построения ЭС на ПЭВМ.

### **3.5. ПиЭС — программный инструментарий для экспертных систем**

*А. Ю. Алешин, В. Ф. Хорошевский, В. Ю. Шерстнев,  
С. Ю. Шенников*

Современные системы автоматизации создания ЭС являются не более чем интегрированными средствами поддержки разработки, ориентированными на знания. Накопленный здесь опыт позволил перейти к следующему этапу автоматизации проектирования ЭС — разработке и реализации инструментальных ЭС, «знающих», как создать сначала «пустую», а затем и прикладную ЭС. Именно такая задача стояла перед разработчиками системы программного инструментария для построения ЭС — ПиЭС.

Впервые концепция инструментальных ЭС была сформулирована в рамках проекта создания системы представления знаний RLL [Greiner et al., 1980]. Однако акцент этой работы был в области создания инструментального расширяемого языка, но не системы в целом. Основное отличие системы ПиЭС от других инструментальных средств заключается в том, что она «знает», как создаются ЭС. Именно эти знания и позволяют системе ПиЭС управлять процессами проектирования и реализации прикладных ЭС [Хорошевский, 1984].

Основными функциональными компонентами любой ЭС являются решатель, ориентированный на знания, собственно база знаний и данных (фактов) и блок «дружелюбного» интерфейса [Хейес-Рот и др., 1987]. Эти процедурно-декларативные компоненты должны быть обеспечены совокупностью процедурных модулей (компиляторов, интерпретаторов, модулей управления и т. п.), составляющих программное окружение. Решатель — исполнительный блок ЭС — включает модуль планирования решения задач (планировщик) в предметной области, для которой предназначена прикладная ЭС, и модуль вывода решения

конкретной задачи, а также библиотеку модулей решения прикладных задач. С решателем тесно связаны блоки данных и знаний, а также рабочее поле, где хранится информация для конкретного сеанса работы ЭС. Однако успех их использования определяется естественностью и «дружелюбностью» интерфейсного блока, поддерживающего общение с пользователем. Исходя из архитектуры развитой ЭС формулируются задачи, которые должен решать пользователь инструментальной ЭС, предназначенной для проектирования и реализации «пустых» и затем прикладных ЭС.

В общем случае правильно организованный процесс создания любой ЭС начинается с разработки мониторинговых блоков. Совокупность их и образует каркас будущей системы, который отличается и от «пустой» ЭС, и от программного окружения. Это лишь «стойка-конструктив», который по мере продвижения разработки заполняется модулями, реализующими основные блоки ЭС. Вместе с тем уже на уровне ЭС-каркаса фиксируются механизмы связи между компонентами будущей ЭС и некоторые архитектурные решения, используемые на уровне как отдельных компонентов, так и системы в целом. Дальнейшее структурирование задачи создания ЭС приводит к следующим подзадачам: разработке компонентов знаний и данных, созданию исполнительного блока и блока интерфейса системы с пользователем. Каждая из этих подзадач, в свою очередь, может быть разбита на более мелкие. Решение каждой из подзадач опирается на знания разработчика конкретной ЭС, и знания эти в большинстве своем отражают его опыт, так как пока нет еще средств строгого формального описания ЭС, а следовательно, и средств автоматической трансляции спецификаций в этой области. Таким образом, инструментальная ЭС — это мощная прикладная ЭС, предметной областью которой является разработка и реализация «пустых» и на этой основе прикладных ЭС. Характерным для инструментальных ЭС является более активное использование процедурных знаний и некоторое снижение «дружелюбности» интерфейса за счет авторизации работы и большей строгости общения.

### Архитектура системы ПиЭС

Основной целью проекта ПиЭС [Хорошевский, 1986] является создание на ПЭВМ типа ЕС-1841, IBM PC/XT и других, совместимых с перечисленными, мощной инструментальной ЭС и развертывание на ее основе серии «пустых» и проблемно-ориентированных ЭС. Комплекс ПиЭС предоставляет пользователю как средства проектирования различных подсистем ЭС, так и инструментальные для создания таких средств. Проектируя «пустую» или прикладную ЭС с помощью системы ПиЭС, пользователь может:

- включать какой-либо компонент инструментальной системы в целевую;
- разработать целевую ЭС, используя специализированные средства высокого уровня;
- создать свои высокоуровневые средства на основе базового инструментария и, используя их, целевую ЭС;
- создать целевую ЭС непосредственно на основе базового инструментария ПиЭС. В ПиЭС в качестве и инструментальной, и рабочих ЭВМ используются ПЭВМ с основной памятью 640 Кбайт, укомплектованные, кроме того, винчестерским диском и накопителем на гибких магнитных дисках, а также монохромным или цветным дисплеем, позволяющим работать в алфавитно-цифровом и графических режимах.

Выбор аппаратной обстановки, естественно, накладывает свой отпечаток и на базовое программное обеспечение. Прежде всего это относится к организации взаимодействия с пользователем на базе модифицированного пакета многооконной графики PEN (исходную версию разработчикам ПиЭС предоставил автор пакета PEN Е. Н. Веселов из ВЦ АН СССР). Пакет PEN поддерживает динамическое определение и переопределение «окон», примитивы текстового обмена с экраном и достаточно мощные графические средства. Базовая операционная система, используемая в ПиЭС, — MS-DOS (версия 3.10 и выше),

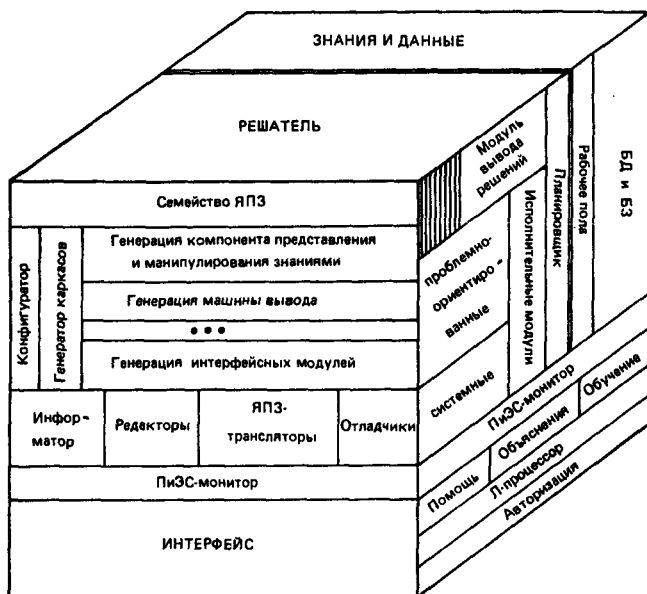


Рис. 3.9. Общая схема системы ПнЭС

а базовый инструментальный язык программирования — Сн. Перечисленные средства программного обеспечения составляют базовый системный инструментарий. В ПнЭС используется также система программирования на базе языка Рефал/2, адаптированная для ПЭВМ [Алешин и др., 1986]. Этот язык, а вернее, его расширение [Khoroshevsky et al., 1987] и является в проекте ПнЭС базовым языком процессора представления знаний и выходным языком трансляторов семейства ЯПЗ.

Общая структура системы ПнЭС представлена на рис. 3.9. Как и в любой современной ЭС, здесь можно выделить три основных компонента: знания и данные, собственно решатель и интерфейс с пользователем. Для организации взаимодействия этих компонентов с операционной системой ПЭВМ, а также для оверлейной подкачки необходимых модулей и нестандартной обработки прерываний служат ПнЭС-монитор. В функции этого монитора входит и начальная установка ключей авторизации, поддерживающих в дальнейшем персонализацию работы, а также администрирование на уровне меню глобальных команд, где перечисляются основные типы деятельности, доступные пользователю.

В рамках ПнЭС-интерфейса выделяют три блока, ориентированные на поддержку различных аспектов взаимодействия с пользователем: авторизация, Л-процессор и модули поддержки общения (модули помощи, объяснения и обучения). Блок авторизации обеспечивает персонализацию общения на основе модели пользователя и авторизованный доступ на базе анализа ключей авторизации и статуса пользователя. К модулю помощи пользователь обращается в тех случаях, когда знает, что хочет сделать, но не знает, как это осуществить, модуль объяснения обеспечивает исследование процессов решения текущей задачи, модуль обучения необходим в тех случаях, когда пользователь не только не знает, как что-то сделать, но и что вообще можно делать в данной ситуации или в рамках всей системы. С модулями поддержки общения тесно связан и последний блок интерфейсной компоненты — Л-процессор. Здесь проис-

ходит собственно анализ запросов и сообщений пользователя, а также синтез вопросов и ответов системы.

Обобщенное взаимодействие с системой ПиЭС осуществляется двумя способами: с помощью иерархических меню (графических и/или текстовых) и через специальное окно ввода-вывода Л-процессора.

Знания и данные в системе ПиЭС структурированы следующим образом. Здесь использована архитектура доски объявлений (blackboard) [Hayes-Roth, 1985], в рамках которой различается управляющая и проблемная память. Вместе эти два вида памяти составляют рабочее поле данных и знаний. В рабочем поле хранятся лишь текущие значения данных, а основной объем знаний и данных находится во внешней памяти и загружается по мере необходимости. Таким образом, рабочее поле как бы продолжается из основной памяти во внешнюю, где переходит в базу данных. Такая организация вызывает необходимость планирования обменов с внешней памятью данными и знаниями, что является функцией включенного в состав решателя планировщика. Проблемы, возникающие при создании модуля вывода решений ПиЭС, во многом те же, что и для ПиЭС-планировщика. Поэтому и ЯПЗ, необходимый для описания стратегии и тактики вывода проблемных решений, единый и для модуля вывода решений, и для ПиЭС-планировщика [Щенников, 1987].

В системе ПиЭС все исполнительные модули делятся на системные и проблемно-ориентированные. В первую группу входят информатор о ходе разработки и совокупность систем программирования для семейства ЯПЗ.

Основными компонентами систем программирования являются языково-ориентированные редакторы, а также интерпретаторы и/или компиляторы входных языков и соответствующие отладчики.

Одним из основных принципов разработки ПиЭС является использование трансляционного подхода к формированию всех уровней будущей системы высшего уровня ЭС-каркаса. Отдельные блоки системы и модули этих блоков описываются с помощью подходящих ЯПЗ, а затем полученные спецификации транслируются в объектные модули. Сборка этих модулей в «пустую» ЭС осуществляется отдельной компонентой — ПиЭС-конфигуратором. На уровне формирования ЭС-каркаса применяется пакетный подход, позволяющий собрать этот блок ЭС в основном из готовых модулей в соответствии со специальным заданием на генерацию, формируемым в диалоге с пользователем. Таким образом, среди проблемно-ориентированных исполнительных модулей выделяются конфигуратор программного продукта, получаемого в рамках инструментальной системы, генератор каркасов разрабатываемых ЭС, а также совокупность подсистем автоматизированного проектирования и реализации отдельных компонентов будущей ЭС. К основным в ПиЭС относятся следующие подсистемы: создание доски объявлений и основных источников знаний для планировщика; разработка компоненты представления и манипулирования знаниями, где интегрируется в будущую ЭС совокупность необходимых ЯПЗ и соответствующих систем программирования; формирование модуля вывода решений и создание блока дружелюбного интерфейса с пользователем.

### Средства представления знаний

В системе ПиЭС используется идея семейства ЯПЗ, каждый из которых ориентирован на свой круг методов представления знаний и требует определенной квалификации пользователей: продукционно-фреймовый язык Пилот и язык расширенных сетей переходов ATNL-2.0 (см. гл. 2). И тот, и другой опираются на уровне реализации на язык символьной обработки Рефал/2, который, в свою очередь, реализован на базовом языке программирования системы ПиЭС — языке Си. Кроме того, в системе ПиЭС инструментальные средства обеспечивают переход от базы данных и знаний во внешней памяти к их эффективному представлению в основной памяти. И, наконец, имеются средства манипулирования данными и знаниями на всех перечисленных уровнях.

Для расширения языка Си средствами работы с фреймподобными примитивами предназначена базовая фрейм-подсистема (БФП). База данных БФП содержит файл прототипов и файл экземпляров. Прототип определяет схему хранения экземпляров в основной памяти, а его слоты описывают возможные типы значений. Экземпляр содержит сами значения в порядке следования слотов в соответствующем прототипе. Синтаксис входного языка БФП определяется следующими форматами:

прототип

```
имя_прототипа || is-a || prototype;
                :
слот-прототипа {слот-прототипа}
```

слот-прототипа

```
имя-слота-прототипа [~[
    || { || | char || ' | } ||
    || ( || | int || | ) ||
    || | || | string ||
    || | || | frame ||
    || | || | func ||
    [значение-по-умолчанию]
```

экземпляр

```
имя_экземпляра || is-a || имя-прототипа;
                :
значение-слота-экземпляра {значение-слота-экземпляра}
```

Таким образом, БФП поддерживает основные типы данных языка Си и, кроме того, множества значений элементарных типов. Множество задается фигурными скобками во внешнем представлении, во внутреннем — данному типу множества соответствуют два значения: число элементов (целое) и ссылка на массив элементов. Отличие фигурных скобок от круглых при определении типа «множество» в том, что во втором случае число элементов множества считается равным числу элементов предыдущего множества. Это позволяет экономить память при работе с разнотипными, но одинаковыми по размеру множествами. Свойство «несохраняемый» (знак «~» перед типом слота) позволяет резервировать место в структурах для значений, которые не поддерживаются БФП. Допустимые значения по умолчанию и значения слотов фреймов-экземпляров полностью определяются их типами.

В основной памяти кроме внутреннего представления фреймов содержатся управляющие таблицы (отдельно для экземпляров и прототипов). В таблице прототипов хранится имя прототипа, ссылка на список слотов и количество экземпляров для данного прототипа. В таблице экземпляров — имя экземпляра, ссылка на внутреннее представление экземпляра, ссылка на прототип данного экземпляра, а также специальные управляющие флаги.

Базовый набор процедур БФП представлен операциями с базой данных; с фреймами и отдельными слотами фреймов. К операциям с базой данных (БД) относятся процедуры установки, чтения состояния текущей БД, загрузки текущей БД в основную память и сохранения ее во внешней памяти, а также очистки и сжатия БД.

К операциям с фреймами относятся: проверка наличия фрейма в памяти; чтение фреймов из БД, удаление фрейма (помимо удаления одного фрейма возможно удаление и всех связанных с ним фреймов); добавление фрейма в таблицу (возможно добавление фрейма с системным именем, неизвестным программисту, что удобно, если доступ по именам нужен не ко всем элемен-

там); поиск фрейма по таблице (при этом доступны процедуры поиска первого экземпляра для данного прототипа и следующего экземпляра для того же прототипа).

Для работы с отдельными слотами экземпляров БФП предоставляет две возможности. Если заранее известна структура экземпляра, можно использовать стандартные средства языка Сн, в противном случае — специальные процедуры БФП для доступа к слотам по имени экземпляра и имени нужного слота. К прототипам применима только одна операция — получение в текстовом виде всех имен и типов слотов.

Разработчику БФП предоставляет достаточно гибкие средства организации фрейм-ориентированных баз знаний (БЗ). Это средства низкого уровня, и они могут использоваться только разработчиками достаточно высокой квалификации. Поэтому подсистема БФП определяет лишь самый нижний уровень средств работы с данными и знаниями в системе ПнЭС, а все средства более высокого уровня доступны из системы РЕБУС (Рефал+База данных+Управление=Система представления знаний).

### Система РЕБУС

В основу системы РЕБУС положен метаалгоритмический язык Рефал (см. гл. 1), вернее, его расширение до современного ЯПЗ, в котором отражены концепция распространения образов на внешнюю память, характерная для языков Пролог и OPS5, а также концепция «прозрачности» управления выводом для пользователя, присутствующая, например, в Интерлиспе.

Обеспечить Рефал средствами работы с базой данных можно несколькими способами. Одним из них является использование готовой системы управления базой данных (СУБД), в которой уже существуют языки описания и манипулирования данными, или соответствующего пакета для работы с данными и знаниями. В первом случае задача сводится к тому, чтобы сделать Рефал включающим языком используемой СУБД, а во втором — к организации перехода от структур данных пакета к спискам Рефала. Для подключения этих компонент к Рефал-системе можно с успехом использовать аппарат машинных операций. Именно этот подход и использован в ПнЭС для подключения пакета БФП к системе программирования на базе языка Рефал/2 для ПЭВМ [Алешин и др., 1986]. Однако при этом выразительные возможности Рефала не меняются. Хотелось бы перенести элегантный производственный стиль обработки структурированных текстов, представляемых в основной памяти списками, на работу с БД, в первую очередь с такой, где объемы и частоты изменений данных и метаданных сопоставимы, а также доступ к данным и метаданным чередуется со сложной невычислительной их обработкой. Поэтому РЕБУС-машина по сравнению с Рефал-машиной имеет еще один вид памяти — базу данных, а ее входной язык является и языком описания данных, и языком манипулирования этими данными. Такой подход требует тщательного конструирования новых изобразительных средств и, кроме того, эффективной поддержки введенных конструкций на уровне реализации соответствующих методов доступа к внешней памяти и резидентного модуля используемой СУБД. Но именно это требует создания системы управления базами знаний (СУБЗ). При этом входным языком такой СУБЗ становится соответствующий ЯПЗ.

Управление в Рефале базируется на жестком порядке вызова функций, определяемом стеком вызовов, и отождествления предложений внутри функций. Стратегия управления в Рефале, заключающаяся в обработке стека вызовов, «защита» в интерпретатор и не может быть изменена. В этой части язык РЕБУС предлагает представление управляющих структур в явном виде за счет введения в язык переменных и спецификаторов новых типов. При таком подходе у программиста появляется возможность планирования порядка выполнения РЕБУС-функций и момента их запуска. Дополнительно к этому в языке РЕБУС появляются естественные средства работы с символами-ссылками с уровня входного языка.

Представление в явном виде структуры управления требует изменений как в декларативной, так и в процедурной части языка. В частности, в РЕБУС вводится понятие иерархического процесса как множества пар процессов, между которыми установлено отношение управляющий-управляемый. На вершине этой пирамиды (нулевой уровень) — системный механизм управления «слева-направо-сначала-вглубь», а в основании — некоторый процесс, который работает на конечный результат (предметный процесс). Кроме того, в языке РЕБУС изменяется представление поля зрения — из линейного списка в многоуровневый, отражающий как вложенность и порядок выполнения РЕБУС-функций на одном уровне иерархического процесса, так и связи между процессами разных уровней.

Наличие в языке мощных средств управления позволяет использовать для обработки различных исключительных ситуаций аппарат прерываний. Прерывание выдает управляемый процесс, а в управляющем процессе описывается соответствующий обработчик прерываний. При этом естественно появляются аппаратные и программные прерывания. Например, ситуация «нет свободной памяти» или «отождествление невозможно» относится к «аппаратуре» РЕБУС-машин и обрабатывается стандартным образом независимо от состояния активного процесса. Обработка ситуаций, относящихся к программным прерываниям РЕБУС-машин, должна быть задана полностью. В противном случае, при неотождествлении (неуспехе) в данном обработчике прерываний, управление получит обработчик прерываний более высокого уровня, а в конечном счете и один из системных обработчиков прерываний.

Пользователем данной СПЗ может быть квалифицированный инженер по знаниям, так как реализация сложных РЕБУС-программ — непростое дело. Вместе с тем именно на этом уровне можно получить практически приемлемую и эффективную СУБЗ.

### Генерация ЭС-каркасов в среде ПиЭС

Одной из главных задач при проектировании любой ЭС является разработка среды, в которой будут функционировать все ее компоненты. Обычный подход к созданию ЭС — концентрация усилий разработчиков на создании модуля вывода решений и тесно связанной с этим блоком СПЗ. Однако данный вариант соответствует восходящей схеме создания системы, трудности использования которой многократно обсуждались в работах по технологии программирования. Основопологающим принципом разработки ПиЭС является нисходящее проектирование и реализация систем. Поэтому одна из первоочередных задач здесь — создание мониторных блоков.

Основной компонент ПиЭС как инструментального комплекса автоматизации проектирования ЭС — блок генерации ЭС-каркасов. Генератор ЭС-каркасов является транслятором, перерабатывающим задание на генерацию в текст программы монитора будущей ЭС. При формировании задания на генерацию возможны пакетный и диалоговый режимы, предпочтителен диалоговый режим, так как при этом пользователь, часто сам того не подозревая, готовит программу ЭС-каркаса на некотором языке спецификаций. Такая программа могла бы быть подготовлена инженером по знаниям и для пакетного режима. Однако при этом возникают дополнительные трудности, связанные с требованиями полноты и однозначности задания на генерацию ЭС-каркаса. Генератор ЭС-каркасов формирует внутреннее представление мониторных блоков, которое, по существу, является БЗ о проектируемом модуле. Далее эта БЗ выступает в качестве задания на генерацию эквивалентной Си-программы, которая обрабатывается «штатным» транслятором.

В ситуации запуска генератора ЭС-каркасов системы ПиЭС на экране визуализируются три окна: визитная карточка ПиЭС; основное текстовое меню, в котором выбрана опция проектирования будущей системы и рабочий стол, где расположено графическое меню, отражающее не только отдельные опции проектирования, но и последовательность их применения. При попытке нарушить технологию разработки система информирует об этом пользователя.



В нашем случае выполнение опции проектирования ЭС-каркаса связано с опросом пользователя по определенному сценарию и формированием БЗ каркаса будущей системы. Фрагмент соответствующего диалога приводится ниже (репликам системы предшествует приглашение вида «—», а ответам пользователя — приглашение вида «⇒»):

— Здравствуйте! Вы находитесь в блоке проектирования каркаса Вашей будущей системы. Управление работой блока осуществляется, как правило, в режиме вопрос-ответ. Для удобства на нижней бровке рабочего стола показаны цвета реплик системы и Ваших ответов. Не забывайте, что в случае затруднений Вы можете обращаться к блокам помощи, объяснения и обучения. А сейчас займемся составлением первичной анкеты на Вашу систему и проектированием головной программы. Введите полное название Вашей системы:

⇒Cattel Consulting System

— Хорошо! А теперь дайте справку по версии Вашей системы. Для этого можно использовать, например, такой формат: Версия Х.ХХ от ДД-ММ-ГГ  
⇒v 0.1 (28-03-88)

— Понятно! И, наконец, назовите организацию и/или лицо, которые будут держателями эталона Вашей системы:

⇒CC AS USSR

— Отлично! В качестве даты начала работы над Вашей системой ПиЭС может взять текущее значение даты с системных часов ПЭВМ. Вы согласны с таким проектным решением (1 — да, 0 — нет)?

⇒ 1

— Все в порядке! Первичная анкета на Вашу будущую систему заполнена. Если Вы устали или не имеете достаточно времени для продолжения работы, целесообразно приостановиться сейчас. Ну, как, отдохнем (1 — да)?

⇒ 0

— И, наконец, более важный вопрос. При входе в систему ПиЭС проверялись Ваши полномочия на работу. Желаете ли Вы, чтобы в Вашей будущей системе тоже был блок авторизации доступа» (1 — да)?

⇒ 1

— Итак, займемся проектированием блока авторизации доступа.

— А теперь спроектируем монитор Вашей будущей системы? Итак, первый вопрос: в системе ПиЭС на экране всегда присутствует ее «визитка». Хотите ли Вы, чтобы этот же метод напоминания об авторах использовался и в Вашей системе (1 — да)?

⇒ 1

— Хорошо! Тогда нам необходимо выбрать место «визитки» на экране и ее графическое представление. Готовы? Нажмите клавишу «ввод».

⇒Enter

— Для удобства работы в каждой системе имеется окно, которое называют «рабочим столом». Мы сейчас займемся проектированием этой части Вашей будущей системы. Обратите внимание, что «рабочий стол» должен быть достаточно просторным и нейтральным по цвету. Готовы? Тогда нажмите клавишу «ввод» и Вы, как обычно, войдете в модуль проектирования окон системы ПиЭС.

⇒Enter

— Обычно перечень основных работ, выполняемых системой, собирается в главное меню. Вы пользовались им для выхода в блок проектирования ЭС, где мы сейчас находимся. Будет ли в Вашей системе аналогичное меню (1 — да)?

⇒ 1

— Хорошо! Тогда займемся проектированием этого меню. Сейчас Вы войдете в специальный модуль диалогового проектирования меню, где сможете вы-

полнить всю необходимую работу. Не забывайте, что при необходимости можно всегда воспользоваться помощью системы ПиЭС. Готовы? Тогда нажмите клавишу «ввод».

⇒ Enter

— Внимание! Мы закончили проектирование мониторной программы Вашей будущей системы. Хотите отдохнуть (1 — да)?

⇒ 1

В результате такого диалога в БЗ текущего проекта формируются фреймы-экземпляры, описывающие каркас будущей ЭС.

Ниже показаны некоторые из них и соответствующие фреймы-прототипы БЗ генератора ЭС-каркасов:

```

info      is_a      prototype;
          sys_name   string;          /* наименование системы */
          sys_vers   string;          /* версия системы */
          . . . . .
syst      is_a      prototype;
          status     int;              /* состояние разработки */
          skeleton   frame;           /* описание ЭС-каркаса */
          probl_solver frame;         /* описание решателя */
          front_end  frame;           /* описание интерфейсов */
          KB_desc    frame;           /* описание БЗ */
skeleton  is_a      prototype;
          status     int;              /* статус ЭС-каркаса */
          main_pr    frame;           /* описание головной */
          monitor    frame;           /* описание монитора */
monitor   is_a      prototype;
          status     int;
          use_TM     int; TM_desc frame;
          wrk_tbl    frame;
          use_glbm   int; glb_menu frame;

proj_inf : info;
«Cattel Consulting Systems»;
«v 0.1 (28-03-88)»;
«CC AS USSR»;

skel_dsc : skeleton; . . . . . mon_dsc : monitor;
                2;                                2;
                main_dsc;                          1; TM;
                mon_dsc;                            wtb_dsc;
                                                1; glbm_dsc;

```

После того как БЗ каркаса системы сформирована полностью, пользователь может сгенерировать по ней объектный и исполняемый модули, соответствующие головной программе будущей системы и ее мониторию блоку. Данная работа выполняется в ПиЭС-конфигураторе, доступном из основного меню системы. При этом вместо всех еще не спроектированных блоков будут автоматически подключены необходимые «заглушки», что позволяет запустить готовую часть на исполнение и убедиться в правильности принятых проектных решений. При необходимости пользователь может перепроектировать отдельные фрагменты ЭС-каркаса или всю эту компоненту, а текущую БЗ сохранить в архиве разработки. Таким образом осуществляется в системе ПиЭС проектирование каркаса будущей ЭС.

### Проектирование исполнительного блока

В графическом меню присутствует опция проектирования блоков будущей ЭС. При ее активации пользователю высвечивается подчиненное текстовое меню, в котором идентифицированы опции «база знаний», «решатель», «интер-

фейс» и «возврат». С точки зрения пользователя проектирование БЗ сводится к выбору формализма представления, описанию структуры БЗ будущей системы и разработке средств ее заполнения. Проектирование решателя связано с выбором и/или разработкой стратегии вывода решений и созданием соответствующего модуля вывода. База знаний и решатель, тесно связаны между собой, и их проектирование должно выполняться, по существу, одновременно. Более того, выбор определенного формализма представления знаний и структур данных на этом уровне в значительной степени определяет синтаксис и семантику решающих процедур.

Необходимо спроектировать ЯПЗ языково-ориентированный редактор и механизм вывода решений, состыкованный с перечисленными компонентами по структурам данных и управлению. Такая работа предполагает высокую квалификацию разработчика будущей системы и наличие мощной инструментальной поддержки. ПиЭС на этом уровне предоставляет пользователю совокупность «готовых» ЯПЗ, механизмов вывода решений и языково-ориентированных редакторов и возможность выбора подходящих средств из заданного набора на основе диалога с пользователем.

На уровне средств представления знаний система ПиЭС поддерживает следующие среды использования языка программирования Си, пакетов РЕН и БФП, использование языка символьной обработки Рефал-2 и его расширения РЕБУС; использование продукционно-фреймового языка Пилот и языка расширенных сетей переходов ATNL-2. Первые два варианта ориентированы на квалифицированных программистов, а остальные — на инженеров по знаниям, имеющих опыт разработки интеллектуальных систем. Все системы программирования обеспечены соответствующими отладчиками и средствами подготовки исходных текстов, которые «имитируются» в проектируемую систему. Однако в настоящее время наиболее удобной для инженера по знаниям является Пилот-среда, на примере которой мы и обсудим технологию заполнения базы знаний систем, создаваемых на базе ПиЭС.

Основу Пилот-среды составляют следующие компоненты языково-ориентированный редактор, Пилот-компилятор, Пилот-библиотека полезных функций, диалоговый многооконный отладчик и средства помощи пользователю. В рабочем столе Пилот-редактора (рис. 3.10) имеется бар-меню с опциями «редактирование БЗ», «компиляция БЗ», «отладка БЗ», «возврат» и три основных окна — для продукционных правил, описания структуры элементов БЗ (прототипы), начального состояния базы фактов (экземпляры). В процессе работы пользователя в рамках первой среды текущего окна правил, прототипов или экземпляров вызывается вспомогательное меню с опциями «взять», «освободить», «очистить», «каталог» и «выход».

Пилот-редактор — языково-ориентированный. В каждом из окон он может работать с представлением вышеуказанной информации в графическом или текстовом режиме. В случае продукционных правил в графическом режиме на экране отображается дерево процедурной части ПИЛОТ-программы, а в остальных случаях — дерево декларативной ее части. И в том, и в другом случае ПИЛОТ-редактор поддерживает работу с определенными структурными единицами. Для продукции это секции и правила, а для прототипов и экземпляров — структуры типа фрейм, слот, значение. Навигация по структурам осуществляется с помощью стрелок. Выделение определенной структурной единицы в графическом режиме вызывает «подсветку» ее инверсным цветом при переходе в текстовый режим. Редактирование типа вставить, удалить, заменить осуществляется на уровне отдельной структурной единицы. Кроме того редактор БЗ отслеживает ситуации неиспользованных и/или неопределенных фактов.

Подготовленная в таком редакторе БЗ может быть скомпилирована и затем запусчена в работу в режиме отладки. При работе с ПИЛОТ-отладчиком пользователь может получать в отдельных окнах трассу выполнения ПИЛОТ-программы и посмотреть значения нужных ему фактов. На этом уровне отладка БЗ ближе к отладке программ, чем к объяснению работы системы для пользователя. Поэтому при «имитации» отлаженной БЗ в прикладную ЭС тре-


Вам доступны ПЛЭС-блоки 1. Учеба 2. Разработка ЭС				
— F1. Помощь — F2. Объяснение — F3. Обучение				
БЗ-сод	БЗ-комп	БЗ-ста	Установка	Возврат
Правила			Экземпляры	
			Прототипы	

Рис. 3.10. Рабочий стол подсистемы накопления знаний в системе ПЛЭС

букета для необходимости подключить специально разработанную подсистему объяснения. Ее проектирование осуществляется в ПЛЭС на уровне создания интерфейсных блоков будущей системы.

Выход решения осуществляется решателем. Стандартная стратегия проектирования этого блока в системе ПЛЭС в настоящее время — выбор одного из вариантов заранее запрограммированных типов решателей. Вместе с тем на этом уровне система может подсказать какой из имеющихся решателей лучше подходит пользователю или посоветовать ему разработать собственный решатель. В последнем случае система может показать программу-заготовку, которую пользователь должен будет отредактировать чтобы получить нужный модуль вывода. Однако чтобы достигнуть модуль вывода с базой знаний на уровне используемых структур данных, необходима довольно высокая квалификация разработчика.

### Проектирование интерфейсов

Успех функционирования прикладной ЭС определяется удобством интерфейсного модуля. Поэтому в системе ПЛЭС проектирование этого блока уделяется серьезное внимание как при создании ЭС-каркаса, так и на уровне разработки основных компонентов будущей системы.

Основу инструментальной поддержки проектирования интерфейсов в системе ПЛЭС составляет меню-администратор. Он предоставляет разработчику интерфейсных блоков простые и удобные средства организации диалога с пользователем, позволяющие ускорить и облегчить процесс проектирования, повысить качество блоков дружелюбного интерфейса выделять проектирование интерфейсов в отдельную работу разделив диалоговые и исполнительные модули, а также обеспечивает модульное проектирование многоуровневых сценариев диалога непосредственно на ПЭВМ, позволяя сразу же исполнять и корректировать различные компоненты сценария. На уровне реализации меню-администратора (МА) были выделены следующие элементы сценария диалога: окно

(определить, показать, убрать, очистить); меню (графические и текстовые); сообщение; сообщение с приемом ответа пользователя; ожидание реакции пользователя; вызов исполнительного модуля.

В качестве базового инструментального уровня меню-администратора используется модифицированный пакет многооконой графики PEN и пакет поддержки интерфейса MAN. Пакет поддержки интерфейса MAN является надстройкой над PEN и поддерживает в многооконом режиме привязку к определенным окнам различных сообщений и картинок, сетевые текстовые и графические меню с произвольной топологией, прием и выдачу форматированных сообщений в любой точке окна. Кроме того, в данном пакете возможно программирование функциональных клавиш.

В структуре меню-администратора выделяются монитор, база данных и знаний, командный процессор и генератор.

Монитор обеспечивает поддержку основных меню и управляет вызовами остальных блоков подсистемы, а также выполнение глобальных команд пользователя, установку режимов МА, наблюдение за состоянием активных баз и трассировку по уровням проектируемого интерфейса.

База данных меню-администратора реализована на основе БФС и хранит текущее состояние процесса проектирования в виде сети фреймов, описывающих сценарий диалога, и информацию, относящуюся к проектируемому модулю.

Генератор предлагает пользователю меню активных модулей (т. е. модулей, находящихся в процессе проектирования) и по его желанию позволяет создать эквивалентные программы на базовом языке программирования системы ПиЭС — языке Си. Непосредственно в среде меню-администратора можно скомпилировать созданные программы и получить исполняемый модуль. Кроме генерации вышеперечисленных процедур пакета MAN и некоторых процедур графического пакета, меню-администратор позволяет вставлять в любую точку проектируемого модуля произвольный текст на языке Си (в этом случае за синтаксическую правильность исходного текста отвечает сам пользователь). При входе в меню-администратор пользователю высвечивается основное меню, которое в дальнейшем появляется на каждом уровне проектирования сценария диалога. По существу, такое меню специфицирует режимы работы меню-администратора на данном уровне. Пользователь может начать или продолжить, если это возможно, проектирование уровня, исполнить созданную часть, удалить уровень (а также все подчиненные уровни) или исправить спроектированный уровень (корректировка, удаление, вставка и исполнение отдельных операторов). Кроме того, на функциональные клавиши «повешена» установка дополнительных режимов (например, включение-отключение трассировки по уровням, очистка экрана перед входом на уровень и др.). Выбрав альтернативу «создание уровня», пользователь попадает в меню возможных операторов, из которого переходит в режим последовательного проектирования данного уровня.

Рассмотрим процесс проектирования уровня на примере меню. Общение с меню-администратором происходит в основном с использованием функциональных клавиш и ответов да-нет. Проектирование меню начинается с определения окна, в котором оно будет существовать. Сначала пользователю предлагается использовать для этой цели текущее окно, а в случае отказа определить атрибуты окна для меню заново. Функциональными стрелками определяются местоположение окна на экране и его цвет. При необходимости вводят текст, который будет находиться на рамке окна, и картинки, которые будут расположены в окне (из списка доступных). Далее пользователь переходит к определению элементов меню. На экране появляется курсор и сообщение «F6 — переход к определению топологии, Esc — отказ от создания меню». Курсор подводится стрелками к нужному месту окна и данная точка фиксируется клавишей «Ввод». Далее появляется меню выбора типа элемента. При выборе графического элемента данная точка помечается, и можно определять следующий элемент меню. Для текстового элемента сначала задается расположение

элемента (вертикальное или горизонтальное), затем стрелками определяется местоположение элемента меню, его цвет. А после этого пользователь входит в мини-редактор для ввода текстов альтернатив. Выбрав все элементы меню, он переходит к описанию связей между ними. Процесс проектирования меню заканчивается указанием элемента, с которого начнется движение по меню, и определением функциональных клавиш.

Переход с одного уровня на другой (например, после проектирования меню верхнего уровня необходимо определить действия под каждой альтернативой и функциональной клавишей) осуществляется следующим образом. Спроектированный уровень необходимо исполнить (для этого используется соответствующая альтернатива основного меню). Попад в меню или другую точку ветвления, можно выбрать любую альтернативу. После этого появляются сообщения о завершении уровня и предлагается выбрать нажатием клавиши одну из следующих альтернатив: перейти на уровень ниже, перейти в начало текущего уровня или вернуться в точку ветвления. Перейдя на уровень ниже, пользователь снова попадает в основное меню, и процесс проектирования может быть продолжен.

Даже краткое описание меню-администратора показывает, что меню-администратор системы ПиЭС — достаточно мощное инструментальное средство проектирования и реализации структуры интерфейсных подсистем различного назначения. Поэтому он может использоваться не только в составе системы ПиЭС, но и автономно, позволяя быстро создавать мониторы общения в диалоге с пользователем. Меню-администратор системы ПиЭС ориентирован на инженеров по знаниям, имеющих опыт программирования диалоговых систем. Если пользователь ПиЭС не может работать на этом уровне, ему предлагается перейти к режиму вопрос-ответ, подобному используемому при проектировании ЭС-каркасов. При этом он неясно, но работает с меню-администратором, однако ведущей в диалоге является система ПиЭС, а не пользователь.

В системе ПиЭС функциональное наполнение сценариев диалога выполняется путем создания процедур обработки ответов пользователя и лингвистических процессоров требуемого уровня сложности. В качестве ЯПЗ на этом уровне может использоваться любой из уже упоминавшихся языков. Вместе с тем в сколько-нибудь сложных случаях для этой цели удобно применить язык ATNL-2.0. Технология его использования похожа на рассматриваемую выше технологию формирования БЗ с помощью ПИЛОТ-среды, а отличается от нее тем, что здесь предлагается «привязывать» отдельные процедуры и(или) блоки Л-процессора к окнам ввода-вывода сообщений, спроектированным в рамках меню-администратора.

Для стандартных случаев в системе ПиЭС имеется библиотека модулей анализа ответов. Чтобы адаптировать их к разрабатываемой системе, пользователь должен дополнить или заполнить заново в соответствующем модуле словарную БД и стыковать семантическую зону словарных статей с предметной БЗ. Эта работа предполагает, что инженер по знаниям взаимодействует со специалистом в области обработки естественного языка. Когда такое взаимодействие невозможно или создается система, не предполагающая использования развитых ЕЯ-средств, в системе ПиЭС предлагается другой вариант разработки интерфейсных подсистем — использование готовых процессоров объяснений и помощи пользователю. Такие процессоры поддерживают лишь определенные типы запросов и рассчитаны на определенные структуры данных. Поэтому проектирование блоков объяснений и помощи с использованием данного подхода сводится к фиксации в режиме диалога необходимых пользователю типов запросов и заполнению БЗ этих блоков соответствующими текстами по заранее сформированным прототипам. Понятно, что с такой работой инженер по знаниям может справиться и без помощи лингвистов. Поддержка этой деятельности в системе ПиЭС осуществляется специальным редактором, стимулирующим разработчика будущей системы к аннотированию всех ситуаций, которые, по его мнению, нуждаются в помощи и объяснениях.

В текущей версии системы ПиЭС при создании отдельных блоков разработчики инструментария иногда вынуждены использовать решения, уже апробированные в системах-оболочках и «пустых» ЭС. Они объединили эти средства в рамках единого комплекса, базирующегося на знаниях о самом процессе проектирования ЭС, что дает уверенность в развитии системы ПиЭС до инструментальной ЭС в полном объеме уже в ближайшем будущем.

## Глава 4.

# Принципы разработки аппаратных средств поддержки интеллектуальных систем

## 4.1. Аппаратная реализация интеллектуальных систем

*В. Н. Захаров, Л. К. Эйсымонт*

### Основные положения

Интеллектуальные системы, или системы обработки информации, основанные на использовании знаний, в последние годы стали распространенным коммерческим продуктом, находящим широкий спрос у специалистов в разнообразных областях. Характерной чертой существующих систем, ориентированных на обработку знаний, является высокий уровень развития их программно-аппаратного обеспечения. При разработке интеллектуальных систем все чаще используются специализированные аппаратные средства, реализующие в той или иной степени основные функции интеллектуальных систем. При разработке универсальных вычислительных систем и их систем программирования часто также учитывается необходимость обеспечения решения задач из области искусственного интеллекта. С целью повышения эффективности работы и быстродействия вычислительных систем (в частности, решающих интеллектуальные задачи) создаются экспериментальные и коммерческие мультипроцессорные ЭВМ с глубоким распараллеливанием процессов выполнения программ [Highberger et al., 1984; Mokhoff, 1984; Hindin, 1984]. Значительный сдвиг в области создания коммерческих мультипроцессорных ЭВМ был сделан в 1985 г.

По типу обрабатываемых данных и используемых операций, а также способам организации вычислительного процесса в интеллектуальных системах можно условно выделить следующие классы задач:

- 1) обработка символьной информации;
- 2) решение переборных вычислительных и логических задач и построение логического вывода решения с использованием заданных систем правил;
- 3) работа с базами данных, содержащих данные со сложными информационными связями;
- 4) высокоскоростная обработка изображений и речи.

Перечисленные задачи в конкретных интеллектуальных системах часто взаимно связаны и предъявляют противоречивые требования к аппаратуре. Например, при работе с базами знаний необходимо обеспечить решение задач символьной обработки, логического вывода, работы с базами данных. Решение каждой из них на машине с традиционной фон-неймановской архитектурой, ориентированной в большей степени на вычислительные задачи, недостаточно эффективно [Fateman, 1978]. Например, высокопроизводительная ЭВМ для обработки числовой информации CDC 7600, решающая задачи обработки символьной информации, работает со скоростью мини-ЭВМ, которая создавалась с учетом решения задач этого класса. Необходимость специализации машин отмечается также в [Boley, 1980; Moldovan et al., 1985].

1. Для задач обработки символьной информации обычно используются языки высокого уровня функционального типа, поэтому при выполнении программ часто происходят обращения к функциям, что связано с большими накладными расходами. Функциональность используемых языков является хорошей базой для распараллеливания программ, но их реализация усложняется из-за применяемых механизмов передачи параметров и доступа к переменным, побочных эффектов при выполнении функций. Для задач этого класса также характерно: применение рекурсивных списковых структур при отображении данных в оперативной и даже внешней памяти; рекурсивный характер обработки списков, при которой используются операции типа поиска в списке и преобразования списка; высокий динамизм использования памяти, управление которой осложняется необходимостью обеспечения работы с виртуальной памятью и работы в реальном масштабе времени. Последнее означает, что любой участок вводимой программы должен выполняться за время, заранее определенное при написании программы. По этой причине недопустимы прерывания процессов обработки информации в произвольные моменты времени для сборки неиспользуемой памяти или для других действий.

2. Переборные задачи, связанные с выполнением логического вывода, включают особенности задач обработки символьной информации, однако для них характерны: большие возможности распараллеливания; частое выполнение операций сопоставления сложных структур данных (это происходит, например, при выполнении процедуры унификации в реализациях языка Пролог); более сложная работа с переменными; более сложная организация схемы управления выполнением программ [Ефимов, 1982].

3. В задачах из области работы с базами данных производится обработка больших наборов данных, размещенных на внешних запоминающих устройствах и имеющих сложные внутренние связи, которые следует учитывать при выполнении операций ассоциативного поиска, выборки и обновления информации. Основная проблема здесь состоит в сокращении времени выполнения этих операций. Различные подходы к организации баз данных (реляционный, иерархический, сетевой) рассматриваются, например, в [Дейт, 1980].

4. При решении задач, связанных с вводом-выводом изображений и речи, необходимо быстрое выполнение большого числа однотипных операций над исходными данными огромного объема. Для этих задач характерна модель группы вычислительных процессов, которые обрабатывают потоки данных, направляя их друг другу по специально организованной сети информационных каналов с определенной структурой (например, модель выполнения быстрого преобразования Фурье).

Из всех перечисленных задач наиболее «необычны» задачи символьной обработки, решаемые при построении интеллектуальных систем для работы со знаниями. Перечислим их основные особенности [Moldovan et al., 1985].

В связи с большой размерностью задач символьной обработки пространство поиска решений очень велико. Для обработки такой информации требуется память большой емкости, а также большая степень параллелизма обработки.

Характерно большое количество обращений к памяти. Без принятия специальных мер разброс близких по времени адресов обращений к памяти достаточно велик. Частичная обработка данных непосредственно в памяти может существенно снизить поток данных.

Данные самоопределены, поэтому перед обработкой приходится сначала распознать их тип. Для этого можно воспользоваться теговой памятью и эффективными специфическими приемами работы с тегами.

Из-за сложной структуры данных, сложной структуры образцов затруднена операция поиска (например, поиск подграфа в графе). Использование обычных ассоциативных процессоров не решает этой проблемы.

Способы вычислений и форматы обрабатываемых данных могут меняться от одной задачи к другой. Архитектура должна допускать настройку на тот или иной случай обработки.



Взаимодействие процессов имеет нерегулярный характер, поэтому нужна гибкость коммутации процессов через информационные каналы.

Алгоритмы часто имеют недетерминированный характер, поэтому распределение вычислительных процессов по процессорам может быть произвольным или производиться уже в процессе работы алгоритма.

Следует также учитывать, что интеллектуальные системы требуют программ большого объема на языках высокого уровня, которые должны эффективно поддерживаться аппаратурой. Кроме того, нужны мощные инструментальные средства разработки программ, т. е. средства программной инженерии. Эта проблема частично решается созданием специальных аппаратных средств, но уже инструментальных, например рабочих станций [Weston et al., 1978; Ohr, 1986a]. О разработке мультипроцессорных рабочих станций см. [Heppessey, 1987; Thacker et al., 1987], об использовании в качестве рабочих станций специализированных машин для обработки символьной информации — [Corley et al., 1985; Atkinson et al., 1987; Hill et al., 1986].

Переход к аппаратной реализации процессов, протекающих в интеллектуальных системах, обеспечивает новые возможности: 1) существенно повышается скорость обработки информации и эффективность отображения в памяти обрабатываемых данных, что позволяет выйти на качественно новый уровень решаемых задач, например к разработке систем принятия решений в реальном масштабе времени; 2) снижается общая стоимость разработки системы за счет использования типовых устройств и элементов; 3) упрощается программное обеспечение системы за счет аппаратной реализации некоторых или большинства интеллектуальных функций; 4) повышается надежность систем за счет повышения устойчивости к сбоям и отказам аппаратуры при реализации программ.

Интерес к исследованиям в области аппаратной реализации интеллектуальных систем возрос после 1982 г., т. е. после того, как Япония выступила с 10-летней программой исследований по ЭВМ 5-го поколения. В США похожий проект был в основном ориентирован на разработку новых систем оружия и получил название «Стратегической компьютерной инициативы» (SCP), хотя кроме чисто военных приложений он направлен и на укрепление экономики США [Highberger et al., 1984; Mokhoff, 1984; Allan, 1986]. Национальные программы по информатике и вычислительной технике, близкие по тематике к разработке технологии и собственно ЭВМ 5-го поколения, были организованы в Великобритании и странах Западной Европы.

Главным показателем качества разработок в области аппаратных функций интеллектуальных систем является повышение производительности машин на 3—5 порядков практически на всех классах задач, решаемых этими системами. Эквивалентная производительность машин нового поколения должна достигнуть сотен миллионов команд в секунду.

Для оценки производительности программно-аппаратных систем обработки символьной информации, которые осуществляют логический вывод, в [Moto-oka et al., 1981] введена единица измерения Липс (LIPS — логический вывод в 1 с). 1 Липс соответствует выполнению 100—1000 команд современных фон-неймановских машин. Похожая единица измерения в СССР используется с конца 60-х годов для оценки скорости выполнения программ, составленных на языке Рефал, — шаг в секунду, где шаг — это выполнение одной Рефал-функции, описанной набором правил подстановок. Производительность современных фон-неймановских машин  $10^3$ — $10^4$  Липс (1—10 КЛипс). Более точные сведения даны в [Dobry et al., 1985], где приведены данные по разным машинам на разных тестовых программах (бенчмарках). Для более простой программы — программы «детерминированной конкатенации» — имеются следующие данные:

VAX 11/780 — 15 КЛипс;  
SUN 2 — 14 КЛипс;  
IBM 3033 — 27 КЛипс;  
DEC 2060 — 43 КЛипс

Для наиболее сложной программы показатель быстродействия заметно уменьшается, например для DEC 2060 он становится равным 12 КЛипс. Для сравнения можно привести оценку производительности ЕС 1045 при выполнении сложных трансляционных Рефал-программ, она равна 2—3 КЛипс.

Распространена еще одна оценка предельной производительности современных фой-неймаховских машин, полученная на скомпилированных Пролог-программах, она соответствует 30 КЛипс [Tick et al., 1984].

В соответствии с планами разработки машин 5-го поколения [Moto-oka et al., 1981] должна быть достигнута производительность 10 000 КЛипс (10 МЛипс) в классе машин символьной обработки без логического вывода (например, Лисп-машин) и 100—1000 МЛипс (до 1 ГЛипс) в классе машин символьной обработки, включающей логический вывод (Пролог-машин).

Близкие оценки по увеличению производительности приводятся и в американских работах. Например, если на современных машинах срабатывание правил экспертных систем происходит со скоростью в несколько сотен в секунду, то для системы управления автоматизированным перевозочным средством сухопутных сил США необходима скорость около 7000 правил в секунду, а для системы управления боем — 12 000 правил в секунду [Allan, 1986]. Одной из целей американской программы Стратегической компьютерной инициативы, руководство которой осуществляет Управление перспективных исследований министерства обороны США (DARPA), — довести скорость обработки сигналов до 1 триллиона (100 млрд.) операций в секунду и увеличить производительность на четыре порядка в классе машин символьной обработки. Этапы этой программы описаны в работе [Mokhoff, 1984].

Можно выделить следующие основные направления работ по архитектуре и языкам программирования машин 5-го поколения:

- накопление знаний о специфике решаемых задач, которая должна проявиться в способах оптимального отображения и организации обрабатываемых данных, в механизмах доступа к переменным, в специальных методах организации вычислительного процесса;

- разработка алгоритмов глубокого распараллеливания процессов выполнения программ на уровне как отдельных участков, так и отдельных операторов;

- разработка архитектур параллельных машин со слабой, средней и сильной специализацией, т. е. создание новых типовых вычислительных элементов таких машин, систем коммутации и систем иерархической памяти;

- определение оптимальных путей использования современных и перспективных технологий электронной техники, прежде всего технологии КМОП-схем, схем на основе арсенида галлия, оптоэлектроники.

Работы по машинам 5-го поколения достаточно полно отображены в материалах Международной конференции, состоявшейся в 1984 г. в Токио и совпавшей по времени с завершением первого 3-летнего этапа японского проекта [FGCS, 1984]. Новые архитектуры машин хорошо представлены также в трудах ежегодных международных конференций по архитектуре вычислительных машин. Особое значение имеет конференция, состоявшаяся в Токио [AISCA, 1986].

Ход выполнения исследований по ЭВМ 5-го поколения неодинаков в разных странах. Например, цель Японского проекта — прежде всего создание технологии, а не конкретных образцов машин, хотя последнее тоже считается важным [Robert, Tate, 1987]. В США цель — конкретные машины, уже имеются хорошие практические разработки вплоть до создания работающих прототипов [Allan, 1986].

Область использования машин 5-го поколения — интеллектуальные системы — слишком молода. Закономерности вычислительных процессов, возникающих при решении интеллектуальных задач, недостаточно изучены. Очень важно практическое внедрение и изучение функционирования в реальных условиях разработанных аппаратных средств, получающихся даже на промежуточных этапах разработки.

Область разработки аппаратных средств для интеллектуальных систем чрезвычайно динамична. Здесь приведены данные, известные к 1988 г. Обзор по аппаратным средствам для систем искусственного интеллекта до 1983 г. имеется в [Вайтершик и др., 1984; Веллер и др., 1984].

### Краткая история

Исследования возможностей построения вычислительных систем нового типа, ориентированных на обработку символической информации и реализации других функций интеллектуальных систем, начались более 50 лет назад.

В настоящее время во всех классах машин ведутся интенсивные поиски возможностей использования параллельных архитектур типа *concurrent flow* — управление потоком команд, *data-flow* — управление потоком данных; *data-driven* — редукционных машин, *symbolic attack* — символьных машин [Gajski et al., 1985].

В области аппаратной поддержки задач обработки символической информации работы велись в различных направлениях: машины с традиционной архитектурой, но с расширенной специальными командами для работы со строками [Sumpter, 1974; Микхордштуу, 1979; Клименко и др., 1981] и со списками [Lewis et al., 1979]. Вводились также команды реализующие специальные алгоритмы обработки [Асратян и др., 1976–1978; Апте et al., 1980]. Разрабатывались машины под языки высокого уровня, которые решали специальные подклассы задач обработки символической информации, например [Глушков и др., 1977].

Перспективной является разработка спецпроцессоров, ориентированных на применение в качестве входных языков специализированных языков высокого уровня, пригодных для программирования широкого класса задач обработки символической информации (см. гл. 26), например таких, как Лисп, Рефал, Пролог. Начат промышленный выпуск машин этого типа. Наиболее популярны Лисп-машины. На первом этапе разработки Лисп-машин важные работы были выполнены в фирме Xerox [Deutsch, 1978, 1980]. Их результаты значительно расширили знания о процессах, происходящих в машине при выполнении Лисп-программ. Очень важны работы по исследованию изменения состояния списковой памяти в процессе выполнения Лисп-программ [Clark et al., 1977; Clark, 1979].

В начале 80-х годов в США был начат промышленный выпуск специализированных машин для обработки символической информации. Появились образцы промышленных Лисп-машин следующих фирм.

Xerox (машини Dorado [Barto et al., 1980; Lampton et al., 1980] и Delphin [Маршал, 1980]), их коммерческие образцы-машины Xerox 1100, Xerox 1105 и Xerox 1132 [Воллер, 1982]).

Bolt Beranek and Newman Inc. (машина Jerce [Маршал, 1980; Greenfield, 1981]).

LISP Machines Inc. (машини LM) типа CADP и машина LAMBDA [Маршал, 1980]).

Symbolics Inc. (машина LM-2 [Маршал, 1980] и машина S3600 [Воллер, 1981]).

В дальнейшем фирма Symbolics стала ведущей, а LISP Machines Inc. разорилась, передав, однако, значительную часть разработок фирме Texas Instruments. Эти разработки нашли отражение в рабочих станциях Explorer [Conley et al., 1985; Мануэль, 1987b], в Лисп-суперкристалле этой фирмы [Matthews et al., 1987] (об этом кристалле см. ниже). Машины фирмы Symbolics семейства S3600 отражают достигнутый уровень в классе символьных процессоров [Moore, 1985, 1987].

Значительное событие — проявление Лисп-кристаллов фирмы Texas Instruments и Лисп-кристалла Ivory фирмы Symbolics [Weste, 1987] (информация об этом кристалле также будет дана позже). Потяжки разработки экспериментальных Лисп-микрпроцессоров делались и раньше [Steele, 1980; Susstap, 1981].

Большинство из существующих Лисп-машин относится к классу CISC (Complex Instruction Set Computer) машин, поскольку их внутренний язык выше уров-

ия системы команд традиционных фон-неймановских машин. Однако наметилась тенденция считать не менее перспективными и спецпроцессоры с сокращенным набором команд класса RISC (Reduced-Instruction-Set-Computer) [Patterson, Sequip, 1982; Patterson, 1985; Hennessy et al., 1982; Hennessy, Gross, 1982, 1983; Hennessy, 1984, 1987; Phsybylski et al., 1984; Sibbey et al., 1986; Gimarc, Milutinovic, 1987] (см. § 4.2).

Первым Лисп-процессором класса RISC является микропроцессор  $\mu 3L$  [Castar, et al., 1982]. Перспективность этих работ оценивается очень высоко, как и вообще направление RISC-машин в целом.

Все Лисп-машини тем или иным образом решают за счет введения дополнительной аппаратуры и специализации четыре основные глобальные задачи: частый вызов функций; организацию доступа к переменным; реализацию операций просмотра и преобразования элементов списковой структуры и их порождения, управление памятью. Варианты решения этих задач удачно изложены в [Pleszkun et al., 1987]. Отметим и другие подходы, характерные для машин типа символьных процессоров в целом, включая Пролог- и Рефал-процессоры, аппаратные реализации продукционных (например, экспертных) систем: оптимизация программ при компиляции [Griss et al., 1981];

выбор наиболее эффективной формы представления программ [Griss et al., 1978];

использование схем экономного выполнения программ (параметры, вызываемые по необходимости [Aiello et al., 1981], задержанное (или «ленивое») выполнение программ [Friedman et al., 1978]);

введение эффективной аппаратной поддержки работы со стеком при вызове функций [Stanley et al., 1987];

параллельное выполнение Лисп-программы [Gusman, 1981; Halstead, 1984, 1985, 1986; Halstead et al., 1986];

конвертирование программ на функциональном редукционном языке, которые хорошо распараллеливаются [Koster, 1980] и для которых не существует трудностей в организации эффективного доступа к значениям переменных;

конвейерное выполнение команд в процессоре, использование значительно большего, чем в обычных машинах, количества общих и «блокируемых» регистров, кэш-памятей, стековой кэш-памяти, специальной памяти для быстрой реализации многоветвенных переходов, эффективной работы с тегами обрабатываемых данных, специальных схем прерываний по программируемым условиям для сокращения количества условных переходов, специальная организация связи арифметическо-логического устройства, сдвиговых регистров и устройства маскирования разрядов [Matthews et al., 1987; Weste, 1987; Steenkiste et al., 1987; Moon, 1985, 1987];

выбор эффективных схем доступа к значениям переменных [Baker, 1978; Pleszkun et al., 1987], являющихся модернизациями двух основных схем привязки значений к переменным, применение лексической локализации переменных, как в языках Scheme и Multilisp [Halstead, 1986];

повышение компактности представления данных в оперативной памяти (относительная адресация Лисп-ячеек с использованием хэш-адресации [Bobrow, 1975] и CDR-кодирование [Bobrow, et al., 1979; Ida et al., 1981], использование векторов при реализации списков [Hansen 1969; Keller 1980; Li et al., 1986]);

обеспечение быстрого доступа к внутренним элементам списков за счет реализации списков векторами, а также за счет специального кодирования элементов списковых структур [Minsky, 1973; Soh! et al., 1985; Potter, 1981; Pleszkun et al., 1986];

специальные алгоритмы, сокращающие количество подкачек листов при работе со списками, размещенными в виртуальной памяти [Fenichel, et al., 1969; Baker, 1978a; White, 1980; Hayashi et al., 1983; Moon, 1984; Yuhara et al., 1986];

разработка алгоритмов распределения памяти, уменьшающих время ожидания повторного использования Лисп-ячейки (наращивающие алгоритмы сборки

мусора [Baker, 1978a], параллельные алгоритмы сборки мусора и использование спецпроцессора для этой цели [Hibilo, 1980; Ram et al., 1986], перенесение некоторой работы по сборке мусора на этап компиляции программы [Ono et al., 1981]), эффективные алгоритмы управления памятью, обеспечивающие работу программ в реальном времени [Moпп, 1984; Wadler, 1976; Liberman et al., 1983].

Ориентация экспериментальных и некоторых промышленных систем обработки символической информации на язык Лисп не является доказательством безуказанности последнего как входного языка для систем искусственного интеллекта. Просто язык Лисп общеприят, и его первые микропрограммно-аппаратные реализации оказались достаточно удачными [Boley, 1980].

В СССР с начала 70-х годов ведутся работы по практическому применению языка Рефал (см. § 6.3) и построению эффективных аппаратно-микропрограммных схем его реализации. Накоплен опыт использования языка для составления разнообразных программ обработки символической информации. Проведено большое количество исследовательских работ в рамках создания эффективной реализации Рефал-процессора [Задыхайло и др., 1975; Эйсмонт, 1977; Задыхайло и др., 1980; Эйсмонт и др., 1982; Головков и др., 1982; Мансуров и др., 1987а, б]. Разработана микропрограммная реализация Рефал-процессора — процессор ЕС2702 [Myamlin et al., 1986; Головков, 1986; Тульский, 1987], который на простых программах развивает скорость 5—7 КЛисп (см. § 6.3). Ведутся работы по реализации новой версии языка Рефал — языка Рефал-4 на мультипроцессорной машине.

Работы по Пролог-процессорам (см. § 6.2) были начаты после объявления японской программы по машинам 5-го поколения. К 1984 г. была реально получена производительность около 30 КЛисп на японской машине PSI [FGCS, 1984]. В других работах сообщается о возможности достижения в ближайшем будущем в последовательных Пролог-процессорах производительности порядка  $10^2$ — $10^3$  КЛисп [Dobry et al., 1985; Tick et al., 1984; Abe et al., 1987; Nakazaki et al., 1986].

Для повышения производительности Пролог-машин применяются многие из перечисленных выше приемов повышения производительности Лисп-машин, а также различные структуры машин (включая data-flow-архитектуры, систолические массивы), которые используют имеющиеся в языке Пролог возможности распараллеливания (OR- и AND-параллелизм, распараллеливание процедуры унификации [Fagin et al., 1987; Shobatake et al., 1986]. Разрабатываются новые варианты языка, которые облегчают параллельное выполнение программ [Sharipo, 1986]. Представляет интерес аппаратная реализация одного из элементов для Пролог-процессоров — процедуры унификации [Woo, 1985], есть даже вариант, в котором используется систолическая структура из однородных простых вычислительных элементов [Shobatake et al., 1986].

Наряду с работами по Лисп, Рефал, Пролог-процессорам ведутся работы по процессорам, реализующим специальные языки программирования экспертных систем, например OPS5, т. е. языкам описания продукционных систем [Gupta et al., 1986; Lehr et al., 1987], а также работы по языкам логического программирования, отличным от языка Пролог, в их аппаратной поддержке [Kohli et al., 1987]. Разрабатываются специализированные параллельные процессоры, например [Moldavan et al., 1985], ориентированный на работу с семантическими сетями, или менее специализированные, но более параллельные машины Connection machine, DADO или NETL, Thistle, Boltzman machine [Fahlman et al., 1983], которые не ориентированы на какой-то язык (см. ниже).

Повышение эффективности работы с базами данных и обработки изображений связано с распараллеливанием операций, повышением быстродействия вычислительных элементов, улучшенном физических характеристик запоминающих сред и пропускных способностей каналов передачи информации. Наиболее типично использование архитектур типа SIMD, MIMD (с множеством потоков команд), а также различного рода систолических структур.

Идея создания машины базы данных возникла в конце 70-х годов [Compter, 1979; Database Eng., 1981]. Работы по созданию специализированных машин

баз данных продолжают (например, [Gajski et al., 1984], работы в рамках японского проекта), однако новый уровень понимания проблемы и сложность технических проблем в последнее время несколько ослабили интерес к созданию машин такого типа [Boral et al., 1983]. Для выполнения операций над базами данных стали использовать более универсальные параллельные процессоры, например Connection Machine [Stanfil et al., 1986] или NON-VON [Hillyer et al., 1986].

Одновременно с созданием аппаратуры для повышения эффективности выполнения операций над базами данных продолжают работы по оптимизации алгоритмов [Database Eng., 1982] и осмысливанию собственно вопросов оценки эффективности, выявления «узких» мест [Database Eng., 1985; Yao et al., 1987].

Для реализации процессоров обработки изображений и речи наиболее характерно использование архитектур типа SIMD (Single-Instruction-Multiple-Data) и MIMD. Первые процессоры для обработки изображений и речи имели достаточно универсальные системы команд [Фостер, 1981]. В дальнейшем специализация этих процессоров усилилась за счет введения специфических процессорных элементов и межпроцессорных связей. Условно их по-прежнему можно отнести к классам SIMD и MIMD. Для этих задач типично большое количество арифметических вычислений и наличие регулярных связей передачи данных между вычислительными процессами. Развитие в этой области идет очень быстро. Наибольшее распространение в этом классе машин получили систолические процессоры [Kung, 1982a; Yen et al., 1982; Kung, 1984].

### Архитектуры ЭВМ новых поколений

Широкое использование сверхбольших интегральных схем (СБИС) и параллельных архитектур повлияло на развитие универсальных машин и на подходы к построению машин специального назначения.

Многие универсальные машины стали, по существу, микропроцессорными системами, причем количество процессоров в них изменяется от единиц до десятков тысяч. Однако по мере увеличения количества процессоров их сложность уменьшается (см. примеры ниже). В мультипроцессорных универсальных машинах применяются различные схемы соединений (коммутации) — от общей шины до схем коммутации типа «гиперкуб» и однородных коммутирующих сред, например омега-сетей. Отметим, что если процессоры таких машин обычно достаточно универсальны, то схемы коммутации, организация оперативной памяти специфичны и ориентированы на класс вычислительных моделей тех или иных задач.

Применение различных вариантов процессоров (или вычислительных узлов), выбор их количества, схемы коммутации зависят от различных факторов, из которых наиболее важные — конкретная область использования, конкурентоспособность машин по стоимости, производительности и другим показателям. Наиболее пригодны для решения задач искусственного интеллекта машины с большим количеством процессоров [Frankel, 1986; Mukhoff, 1987; Bond, 1987; Мануэль, 1987б]. В качестве примеров рассмотрим наиболее часто упоминаемые мультипроцессоры, которые либо выпускаются промышленностью, либо разрабатываются в исследовательских центрах ведущих фирм.

Машина Connection Machine фирмы Thinking Machine содержит 65 536 однокорпусных процессоров, они объединяются сетью типа «гиперкуб», производительность машины до 1 млрд. опер./с [Frankel, 1986; Mukhoff, 1987].

Машина Butterfly фирмы Bolt, Beranek and Newman содержит до 256 процессоров 68020, ее производительность до 200 млн. опер./с. Процессоры соединены через матричный коммутатор.

Машина MPP фирмы Good Year Aerospace содержит 16 384 однокорпусных процессора, объединенных в виде матриц. Машина продемонстрировала производительность 6,5 млрд. опер./с (операции типа сложения с фиксированной точкой).

Машина RP3, разрабатываемая в исследовательском центре фирмы IBM совместно с академическими институтами, будет содержать до 512 процессорных элементов, каждый из них выполнен на базе RISC-микропроцессора ROMP фирмы IBM, связанных через быструю общую шину и разделенных на 8 групп по 64 элемента. Производительность машин будет до 1 млрд. опер./с.

Машины iPSC/2 и iPSC/2-VX (векторный вариант) выпущены фирмой Intel Scientific Computers [Мануэль, 1987]. Вычислительный узел машин содержит микропроцессор 80386 и сопроцессор плавающей точки 80387 (векторные варианты содержат еще процессор обработки векторов), оперативную память емкостью 1, 4, 8 или 16 Мбайт, коммутатор прямых соединений каналов передачи информации между вычислительными узлами, кэш-память. В машине может быть от 32 до 128 таких узлов. На задаче двумерного быстрого преобразования Фурье компьютер iPSC/2-VX с 32 узлами показал производительность 154 млн. Флопс (операций с плавающей точкой в секунду), а по оценочной программе «Треугольник Габриэля», написанной на языке Лисп, — в 4 раза большую производительность, чем компьютер Cray-1S. Таким образом, iPSC/2 пока является одновременно и самой быстрой Лисп-машинной в мире.

Еще один класс универсальных машин, важных для разработок систем искусственного интеллекта, — высокопроизводительные рабочие станции [Электроника, 1980; Thacker et al., 1987; Ohr, 1986a; Hennessy, 1987; Atkinson et al., 1987]. Их производительность оценивается в 10—100 млн. опер./с; производительность (VAX 11/780) 1,1 млн. опер./с. Такие станции уже конкурируют со специализированными машинами при решении задач обработки символьной информации [Verity, 1986; Meng, 1986].

Однако практика построения реальных интеллектуальных систем и их применения показала необходимость эффективной поддержки самых различных функций в рамках одной системы. Например, реальные задачи обработки символьной информации требуют высокой скорости обработки символьной информации и достаточно быстрого выполнения вычислений [Thomas, 1987]. Это уже нашло отражение в архитектуре некоторых процессоров обработки символьной информации, например SM45000 фирмы Integrated Inference Machines [Мануэль, 1987], промышленная Лисп-машина Symbolics 3600 также имеет устройство ускоренного выполнения операций над числами с плавающей точкой [Moon, 1987]. Известно о применении Лисп-станции Explorer фирмы Texas Instruments как фронтальной машины, подключенной к супермашине, выполняющей расчеты. В этом случае Лисп-станция производит оптимизацию вычислительного эксперимента, происходящего на супермашине [Bate, 1987]. Создают также алгоритмы обработки сигналов, включающих совместное использование традиционной числовой обработки и обработки символьной информации [Bate, 1987]. «Численно-символьные (аналитические)» алгоритмы используются при решении задач математической физики [White, 1987].

Необходимость интеграции специализированных средств решения задач искусственного интеллекта, как и вообще специализированных устройств, в рамках одной вычислительной системы известна давно [Lecht, 1977; Rescoque, 1980; Марчук и др., 1980; MBC, 1975]. Однако в реальных системах это вводится с запаздыванием. В мультипроцессорных универсальных машинах такая «интеграция» уже есть, по крайней мере задачи различных классов можно решать на разных процессорах.

В ряде выпускаемых промышленностью мультипроцессорных систем и рабочих станциях используются микропроцессоры с усовершенствованной архитектурой фон-неймановского типа (RISC-процессоры). Все ведущие промышленные фирмы или уже выпускают RISC-процессоры, или готовят их выпуск [Gimarc et al., 1987; Электроника, 1987]. Ясно, что процессоры с RISC-архитектурой не решают всех проблем, в некоторых случаях могут оказаться более выгодными CISC-процессоры [Silbey et al., 1986; Мануэль, 1987] либо использование одновременно элементов CISC- и RISC-процессоров [Mokhoff, 1986; McNeby et al., 1987]. Полемика по поводу CISC- и RISC-архитектуры ведется до сих пор [Джой, 1987; Смит, 1987], хотя уже можно утверждать, что RISC-архитектура

стала наиболее популярной, в частности, для новой технологии разработки быстродействующих интегральных схем на основе арсенида галлия (GaAs) [Milutinović et al., 1986a, b].

Работы по GaAs-микропроцессорам требуют более подробного рассмотрения, поскольку в перспективе они могут определить развитие вычислительной техники. Ставится задача разработки 32-разрядного микропроцессора с производительностью около 100 млн. опер./с (тактовая частота 200 МГц).

Разработка GaAs-микропроцессоров с RISC-архитектурой производится [Karр et al., 1986; Барни, 1985] для построения высокоскоростного процессора обработки сигналов [Gilbert et al., 1986; Naused et al., 1987]. Рассматривается также вариант процессора для использования в системах реального времени [Milutinović et al., 1987b].

При разработке GaAs-микропроцессоров за основу была взята архитектура RISC-микропроцессора MIPS, разработанного в Стэнфордском университете под руководством Хеннесси [Przybylski et al., 1984]. Этот микропроцессор в несколько измененном виде выпускается промышленностью [Ohr, 1986; Weiss, 1987] и называется R2000. В Стэнфордском университете создан новый вариант микропроцессора MIPS-X с тактовой частотой 20 МГц [Chow et al., 1987]. R2000 и MIPS-X разработаны с использованием КМОП-технологии. Варианты GaAs микропроцессоров рассматриваются в [Fox et al., 1986; Rasset et al., 1986; Milutinović et al., 1987b]. Проблемы разработки компиляторов для GaAs-микропроцессоров с RISC-архитектурой рассмотрены в [Milutinović et al., 1987a], вопросы интеграции GaAs-схем на одной пластине — в [McDonald et al., 1987].

Микропроцессоры RISC и другие 32-разрядные микропроцессоры с усовершенствованной фон-неймановской архитектурой [Atkinson et al., 1987; Ditzel et al., 1987; Pleszkun et al., 1987] можно рассматривать как новые «строительные блоки» машин будущего. Наиболее ярко это выразилось в микропроцессоре Transputer фирмы Inmos [Whitby-Strevens, 1985; Barron и др., 1983; Mattos, 1984; Roelof, 1987a, b]. Фирма Inmos представляет этот микропроцессор как аппаратную реализацию языка параллельного программирования Оккам [Hull, 1987] (см. § 4.2), который, в свою очередь, опирается на определенную идеологию построения параллельных программ и вычислительных систем [Wilson, 1983a; Curry, 1984]. В работах [Frankel, 1986; Computerwold, 1986] описывается построенная на микропроцессорах Transputer и процессорах фирмы Weitek мощная вычислительная система серии T фирмы Floating Point Systems (FPS) для решения вычислительных задач с пиковой производительностью 262 ГФлопс. В построенной на транспьютерной мультипроцессорной машине Computing Surface фирмы Meiko Ltd (Великобритания), которая будет в первую очередь использоваться для обработки изображений, достигнута пиковая производительность 3 млрд. опер./с [Электроника, 1986]. В работе [Cencalves et al., 1988] сообщается о микропроцессоре для обработки символьной информации, тоже построенном на транспьютерах.

Достижения в области разработки мультипроцессорных вычислительных систем повлияли на создание мощных микропроцессоров и схем коммутации, а также на создание кэш-памятей, блоков управления памятью, схем синхронизации. Обзор различных подходов к построению блока управления памятью в виде сверхбольшой интегральной схемы имеется в [Furht et al., 1987]; различные варианты организации кэш-памятей рассматриваются в [Goodman, 1987; Dobojs et al., 1988]. Схемы синхронизации в мультипроцессорной системе приведены, например, в [Beck et al., 1987; Dobojs et al., 1988]. Исследование различных областей применения не ослабило актуальности разработок специализированных вычислительных машин. Там, где необходимо получение наивысшей производительности с меньшими аппаратными затратами или параллельная производительность, разрабатываются специализированные мультипроцессорные системы со специальными вычислительными узлами и схемами их коммутации.

В классе машин обработки символьной информации разработаны промышленные Лисп-процессоры, которые в дальнейшем будут использоваться при построении высокопараллельных специализированных машин. В настоящее время



они нашли применение в промышленных образцах последовательных Лисп-машин.

На кристалле Лисп-микропроцессора размещается более 60 % схем двух-платного центрального процессора рабочей станции Explorer, вместе с тем по вычислительной мощности Лисп-микропроцессор в 5 раз превосходит этот центральный микропроцессор.

Архитектура Лисп-микропроцессора фирмы TI не сильно отличается от центральных процессоров станций Explorer. Это микропрограммно управляемый микропроцессор с регистровой, блокнотной и кэш-памятью большой емкости. Используется внешняя управляющая память микрокоманд ( $32K \times 64$  бит), а также небольшая внутрикристалльная память микрокоманд ( $256 \times 64$  бит). Около 80 % площади кристалла занято памятью для хранения данных и микрокоманд, ее емкость — 114 Кбит (приблизительно 3,5 тыс. 32-разрядных слов). Используется шесть типов памяти:

- 64×32 бит — малая блокнотная память;
- 1K×32 бит — большая блокнотная память;
- 1K×32 бит — кэш-верхушки стека;
- 2,5K×18 бит — память для хранения таблиц адресов микрокоманд, она позволяет осуществлять многоветвенные переходы (до 128 ветвей) с такой же скоростью, как и обычный одно-ветвенный переход;
- 64×32 бит — стек адресов микропрограмм;
- 256×64 бит — внутрикристалльная микропрограммная память для хранения микропрограмм начальной загрузки и тестирования.

Имеется блок преднаказки команд микропроцессора с буфером в четыре 16-разрядных слова. Арифметическо-логическое устройство 32-разрядное, есть 32-разрядный регистровый сдвигатель, сдвигатель регистр, блок маскирования разрядов. Остальные схемы — управление, буферные памяти.

Вместе с Лисп-кристаллом был разработан отдельный кристалл управления трансляцией виртуальных адресов и буферизации данных (MMU — Memory Management Unit) [Matthews et al., 1987].

Лисп-микропроцессор TI будет применяться в бортовых управляющих системах [Вулф, 1987; Amundsen et al., 1985]. Намечается также его использование в рабочих станциях фирмы TI [Computer Des., 1987].

Лисп-микропроцессор Ivory фирмы Symbolics — результат переработки архитектуры Лисп-машин, которые фирма Symbolics изготовила ранее. Здесь заметно влияние RISC-идеологии. Ivory — это 40-разрядный микропроцессор (32 разряда — данные, 8 разрядов — теги), имеющий около 200 команд, наиболее часто использовавшихся в Лисп-машине прежней архитектуры. Команды 16-разрядные. Большинство команд выполняется за 1 такт, тем не менее они разные по мощности: от команд ADD (сложить), SUB (вычесть), PUSH (занести в стек) до команд, реализующих встроенные функции MEMBER и UNIFY (унификация для реализации языков логического программирования). Микропроцессор Ivory конвейерный, число ступеней конвейера равно четырем. В микропроцессоре Ivory имеется арифметическо-логическое устройство, совокупность циклических сдвиговых регистров, блок маскирования, блок обработки тегов. Поддерживаются возможности интенсивной работы с памятью (возможность одновременно работать с тремя потоками данных из памяти), есть средства поддержки работы в мультипроцессорной системе, средства поддержки объектно-ориентированного программирования.

Для разработки высокоскоростных специализированных микропроцессоров многие разработчики пытаются применить архитектуру типа RISC, расширенную операциями работы с тегами [Patterson, 1987; Hill et al., 1986; Taylor et al., 1986]; а также просто высокоскоростные RISC-микропроцессоры [Steenkiste et al., 1986]. Однако в [Steenkiste et al., 1987] показывается, что для эффективного

выполнения Лисп-программ специализация RISC-микропроцессора все-таки нужна.

Делались попытки реализации Пролога на микропроцессоре типа RISC [Borgiello et al., 1987], ориентированном на эффективную поддержку выполнения Лисп-программ. Результаты оказались достаточно хорошими, однако выявилась необходимость определенной специализации.

Есть работы, в которых реализации специализированных программно-аппаратных систем используют стандартные аппаратные блоки, работающие на очень высоких частотах. Например, Проект LOW RISC ставит своей целью реализацию сверхбыстрого процессора языка Пролог на очень простых микропроцессорах с RISC-архитектурой, но с использованием высокочастотных электронных технологий (ЭСЛ, арсенид галлия), которые не обеспечивают большой интеграции на кристалле [Mills, 1987; Short et al., 1987].

Обзор работ по Лисп-машинам с обширной библиографией имеется также в работе [Мямлин и др., 1988].

При проработке отдельных микропроцессорных элементов исследовалась поддержка не только языков Лисп и Пролог, но и объектно-ориентированных языков. Здесь можно выделить исследовательский специализированный RISC-микропроцессор SOAR [Ungar et al., 1984]. Схемотехнические решения микропроцессора SOAR в дальнейшем были использованы в RISC-микропроцессоре Sparc фирмы Sun Microsystems, являющемся основой рабочей станции SUN-4/260 [Электроника, 1987].

Ведутся исследования промышленных специализированных микропроцессоров для обработки символьной информации. Например, разрабатывается рабочая станция SPUR (Symbolic Processing Upon Risc), которая должна содержать 6—12 специализированных RISC-процессоров. Каждый процессор будет иметь свою кэш-память и подключаться через общую шину к разделяемой общей памяти [Patterson, 1987]. Эквивалентная производительность такой рабочей станции ожидается около 50 млн. опер./с.

Продолжается работа по машинам 5-го поколения в Японии. Заканчивается вторая часть этого проекта. На этом этапе планируется объединение машины логических выводов PSI [Taki et al., 1987] и машины баз данных Delta [Murakami et al., 1983; Kakuta et al., 1985], построенных на первом этапе (к 1984 г.), в единую машину базы знаний.

Для обработки сигналов и изображений разрабатываются специализированные системы, например систолический процессор WARP [Anaratone et al., 1986] или матричный процессор AAP2 [Kouido et al., 1986], специальные процессоры для быстрого преобразования Фурье [Hasegawa et al., 1986] и др. Процессоры этого класса выпускаются в различных вариантах промышленностью [БИНТИ ТАСС, 1988б].

Работы в области новых архитектур типа data-flow не дают пока значительных практических результатов, хотя по-прежнему считаются перспективными [Veen, 1986; Shimada et al., 1986].

### Перспективы и прогнозы

В дальнейшем основной рост производительности в области аппаратных средств будет обеспечиваться за счет повышения производительности отдельных вычислительных элементов (специализация, элементная база), а также за счет мультипроцессорных структур типа MIMD, в которых по-прежнему сохранится управление потоком команд (control flow), как и в машинах фон-неймановского типа. Скорее всего наиболее распространенными будут мультипроцессорные системы, включающие специализированные и универсальные RISC- и CISC-микропроцессоры, которые соединяются либо быстрой шиной, либо матричным коммутатором, либо сетью с передачей сообщений или непосредственной коммутацией каналов и работают с общей памятью и/или распределенной локальной памятью.

Для усовершенствования вычислительных элементов таких систем важное практическое значение имеет разработка RISC-процессоров на арсенале галлия и компиляторов для них. Основная проблема в разработке компиляторов здесь связана с необходимостью конвертации кода, так как процессоры на схемах арсенала галлия будут конвейерными с большим числом ступеней конвейера.

Проблема дальнейшего повышения производительности при обработке произвольной информации тесно связана с проблемой рациональной архитектуры создаваемых вычислительных систем. Разработка машин с управлением потоком данных является одним из основных перспективных направлений мультипроцессорных систем, в котором довольно успешно преодолеваются ограничения традиционной фон-неймановской архитектуры. В машине с управлением потоком данных реализуется тот естественный параллелизм, который присутствует в методе решения данной задачи. Тем самым достигается предел возможного параллелизма в алгоритме ее решения. Теоретической основой для построения модели графа потока данных может служить аппарат сетей Петри [Petri, 1973], наглядно отображающий динамику вычислений. Из специальных языков программирования для машин потоков данных наиболее известными являются языки Id Калифорнийского университета в Ирвине [Arvind et al., 1978] и VAL Массачусетского технологического института [Ackerman et al., 1979], обзор языков дан в [Ackerman, 1982].

В лаборатории по вычислительной технике MIT под руководством Арвинда строится 64-процессорная data-flow-ЭВМ на основе модели маркировщика токенов, которая позволяет динамически приспособлять структуру ЭВМ к вычислительному алгоритму. В прототипе используются заказные СБИС [Agervall et al., 1982; Arvind et al., 1980].

Начав исследования в области функциональных языков программирования в фирме Bittoughs, А. Дейвис продолжает их в Университете штата Юта. Архитектура его вычислительной машины ориентирована на язык программирования Лисп и на внутреннее управление ЭВМ типа demand-driven [Davis et al., 1982].

В Лаборатории вычислительных машин при Университете в Манчестере работает одно кольцо data-flow-ЭВМ. Конечный вариант должен содержать четыре кольца. В этой ЭВМ применяются маркированные токены, как и в ЭВМ Арвинда [Gurd et al., 1980a, b; 1985], опубликованы результаты исследований работы этой машины [Gurd, 1983; Gurd et al., 1983], варианты ее развития [Patlaik et al., 1986]. Это наиболее развитый проект data-flow-машины. Основная проблема здесь, как и в других data-flow-машинах — большие накладные расходы при выполнении вычислений. В ряде работ делаются попытки ликвидировать такой недостаток за счет ограничения data-flow-механизма, объединяя его с управлением от потока команд [Sowa, 1987].

В Японии data-flow-организация вычислений является одним из основных принципов в проекте архитектуры вычислительных машин 5-го поколения. В настоящее время разработаны две data-flow-архитектуры. TOPSTAR — в Токийском университете (уже налажено серийное производство) и проект фирмы Nippon Telephone and Telegraph Corporation.

В социалистических странах по проблеме data-flow-организации ЭВМ и по другим, близким к ней проблемам уже много лет ведутся научные работы. В СССР проводятся исследования в области рекурсивных ЭВМ, а также разрабатываются модели параллельных вычислений на базе сетей Петри. Представляют интерес исследования многопроцессорных архитектур систем распределенной обработки данных и организация распараллеливания процедур логического вывода в таких системах.

Несмотря на обилие исследований использование архитектур типа data-flow [Veep, 1986], data-driven, редуцированного типа [Vegdail, 1984] по-прежнему остается проблемой, хотя и стали появляться отдельные образцы коммерческих data-flow-машин [Kariat, 1987] и даже специализированные микропроцессоры, например FDT281 для обработки изображений [Meshach, 1984].

Интересны исследования и уже появившиеся коммерческие образцы машин

с горизонтальной архитектурой, или машин со сверхдлинными командами, которые специалисты называют такие машины статическими [Cohew et al., 1986] и отметить появившиеся коммерческие машины семейства Sparc фирмы Sun Microsystems [Cohew et al., 1987].

Интересны исследования в области электронно-оптических систем. Хотя ведутся уже около 20 лет и было немало прогнозов относительно создания статических машин, которые не оправдались, работы последнего времени [Betta et al., 1987; Guha et al., 1987] свидетельствуют о значительных успехах и скором появлении таких машин.

В последние годы стали вестись исследования по «сверхбыстрым ЭВМ», их перспективы оцениваются высоко [БИНТИ ТАСС, 1988а].

Произошли серьезные сдвиги и в области архитектуры вычислительных машин. появилось множество коммерческих мультимикропроцессорных машин сверхвысокие микропроцессоры, качественно улучшились элементная база и системы автоматизации проектирования. Произошла интеграция усилий различных фирм и университетов по созданию новой техники и программного обеспечения в рамках хорошо организованных национальных проектов и в многочисленных государственных исследовательских центрах. Все это резко повысило темпы выполнения исследовательских и опытно-конструкторских работ, так что возможность достижения целей, поставленных в начале 80-х годов по машинам 5-го поколения и развитию вычислительной техники, не вызывает сомнений.

## 4.2. Элементная база интеллектуальных систем

*В. Н. Захаров, Л. К. Эйсымонт*

### Структура элементной базы

Одной из основных проблем построения аппаратного обеспечения высокопроизводительных вычислительных и управляющих систем всегда был выбор соответствующей элементной базы и применяемых типовых устройств и модулей, удовлетворяющих различным требованиям: надежности, быстродействию, стоимости, энергопотреблению, компактности и т. п. Разработка аппаратного обеспечения интеллектуальных систем также невозможна без успешного решения этой проблемы.

Учитывая высокую сложность систем, ориентированных на работу со знаниями, различное функциональное назначение входящих в них блоков, разнородность обрабатываемых данных и знаний, можно выделить следующие три уровня организации аппаратной поддержки протекающих в них процессов.

Первый уровень — модульный, или общесистемный. Элементная база — крупные, сложно организованные модули в виде реализованных «в металле» специализированных машин, процессоров, контроллеров, интерфейсных систем и других устройств, которые можно рассматривать как типовые блоки при построении интеллектуальных систем. Реализация «в металле» типовых устройств общесистемного уровня осуществляется, как правило, из отдельных микросхем, размещаемых на платах печатного монтажа.

Второй уровень — микросхемный. Элементная база — однокристалльные машины, специализированные процессоры, устройства памяти, вычислительные модули микропроцессорных систем, выполненные в габаритах отдельных микросхем. С развитием микроэлектронной технологии и повышением уровня интеграции наметилась тенденция перевода устройств, относящихся к элементной базе первого уровня, в микросхемы элементной базы второго уровня. Эта тенденция сохранится и на последующем этапе развития вычислителей новых поколений.

Третий уровень — микроэлектронный. Элементная база — конкретные схемы (базовые узлы микросхем), реализующие основные операции булевой алгебры

(И, ИЛИ, НЕ, И — НЕ, ИЛИ — НЕ, НЕРАЗДЕЛИТЕЛЬНОЕ ИЛИ, ЛОГИЧЕСКАЯ РАВНОЗНАЧНОСТЬ и др.), а также операции, реализующие функции задержки или памяти (схемы триггеров, сдвиговых регистров и т. п.). На способ построения всех этих схем существенное влияние оказывают: технология производства микросхем (биполярная или КМОП-технология); вид используемых материалов (кремний, кремний-на-сапфире, арсенид галлия); способы синхронизации и контроля работоспособности (схемы синхронные, асинхронные, самосинхронные, или апериодические) и степень интеграции элементов, размещаемых на кристалле, определяемая чаще как  $k$ -микронная технология, где  $k$  — минимальный размер элемента транзистора, мкм. При проектировании собственно микросхем иногда выделяют следующий, более детальный, элементный уровень, различая схемные и конструктивные элементы интегральных схем (ИС). Так, для интегральных схем, использующих в качестве базовой структуру металл — окисел — полупроводник (МОП), схемными элементами являются транзистор, конденсатор, резистор, а конструктивными элементами — затворный диэлектрик, коммутационная шина, межслойный контакт и др. Таким образом, конструктивный элемент в масштабе микросхемы — это специальным образом сформированная в объеме или на поверхности полупроводника область с определенными электрофизическими свойствами. Любой схемный элемент может быть представлен совокупностью отдельных конструктивных элементов. Выделение в ИС конструктивных элементов производится в предположении их технологической независимости при изготовлении [Брагин и др., 1988].

Типовые устройства первого уровня организации аппаратной поддержки интеллектуальных систем можно разделить условно на три типа: специализированные машины для эффективной реализации языков программирования высокого уровня, для управления интеллектуальными базами данных и базами знаний и для поддержки интеллектуального интерфейса.

Наиболее распространенными устройствами первого типа являются машины поддержки таких традиционных для интеллектуальных систем языков программирования высокого уровня, как Лисп, Пролог, Рефал (соответственно Лисп-машины, Пролог-машины, Рефал-машины), а также появляющихся новых языков функционального программирования. Подробные сведения об особенностях перечисленных машин приведены в гл. 6.

Машины для интеллектуальных баз данных и баз знаний можно условно разделить на четыре группы: машины для баз данных и знаний, основанные на представлении знаний в виде семантических сетей; продукционного типа, основанные на представлении знаний в виде систем продукций; для объектно-ориентированных баз данных и знаний; для баз данных и знаний, основанных на моделях, имитирующих нервные сети.

Такое деление весьма условно и отражает выделенные направления исследований в области разработки машин для интеллектуальных баз данных и знаний, а не конкретно реализуемые проекты.

В отдельную группу выделяются машины логического вывода, реализующие алгоритмы дедуктивного, индуктивного и правдоподобного вывода и входящие в качестве составных блоков в машины баз данных и знаний.

Наконец, к специализированным средствам поддержки интеллектуального интерфейса относятся машины для распознавания речи, образов и обработки изображений, системы машинного зрения.

Многие из перечисленных выше машин реализованы частично или полностью программным путем. Построение эффективных средств аппаратной поддержки для многих машин — нерешенная проблема.

Одним из способов организации аппаратной поддержки интеллектуальной системы, реализуемой программным путем чаще всего на каком-либо персональном компьютере, является подключение к этому компьютеру специально разработанного дополнительного блока. Этот блок позволяет повысить производительность компьютера, решающего интеллектуальную задачу, за счет ускорения выполнения таких операций и процедур, как компиляция с языков высокого уровня, формализация задания, обработка текста или речи, поиск информации

в базе данных или знаний, построение графических изображений в реальном масштабе времени и т. п. При использовании таких блоков появляется возможность решения интеллектуальных задач, требующих очень высокой производительности, которая не может быть обеспечена выбранным персональным компьютером. В качестве примера можно указать на разработку некоторыми фирмами (в частности, INMOS) дополнительных блоков, повышающих возможности одного из самых распространенных персональных компьютеров IBM PC.

Появились многопроцессорные (мультипроцессорные) системы на основе типовых блоков, дающие возможность создавать системы распределенной обработки информации произвольной конфигурации. Основным строительным блоком при создании таких систем часто служит специальная микросхема, называемая транспьютером, которая является представителем элементной базы второго уровня.

С функциональной точки зрения элементная база второго уровня для интеллектуальных систем полностью еще не определена. Решению этой задачи будет способствовать развитие работ в области исследований возможностей аппаратной реализации процессов в интеллектуальных системах путем создания реальных физических макетов. В связи с этим возникает задача разработки базового модуля (элемента) как основы построения универсальных макетирующих сред или мультипроцессорных систем, имитирующих процессы обработки информации в интеллектуальных системах. В общем случае такой элемент должен обладать следующими свойствами [Глухи и др., 1987].

1. Базовый элемент как элемент высокопроизводительной вычислительной среды должен быть универсальным программируемым преобразователем дискретной информации, т. е. его функции должны быть не менее богаты, чем функция микроЭВМ. Требование следует из того, что для решения задач более широкого класса, чем просто логические или вычислительные, полная система операций преобразования заранее неизвестна.

2. Базовый элемент должен иметь развитые интерфейсные средства, обеспечивающие доступ к памяти непосредственных соседей, участвующих в совместном преобразовании (обработке) некоторой порции данных.

3. Базовый элемент должен иметь многосторонние средства приема и передачи дискретной информации (например, по четырем направлениям в соответствии с числом возможных связей с ближайшими соседями в двумерной однородной среде). Средства связи должны быть универсальными (дуплексными с возможностью последовательной и параллельной передачи данных). Элемент должен обладать памятью для хранения данных, принятых по любому из направлений, до тех пор, пока он не сможет приступить к их обработке (т. е. пока занят центральный процессорный элемент).

4. Средства связи базовых элементов должны обеспечивать возможность соединений элементов по входам и выходам (в том числе соединений с шинами).

5. Базовый элемент должен иметь развитые средства согласования взаимодействия (например, типа хендшейка).

6. Элемент должен быть спроектирован с учетом требований специализированного языка программирования (служить ему эффективным аппаратным средством поддержки), ориентированного на реализацию программ в сети распределенной обработки данных.

Транспьютер в основном соответствует перечисленным требованиям и поэтому пригоден в качестве базового элемента для универсальной макетирующей среды, особенно если учесть, что транспьютер является основой для аппаратной поддержки специального языка программирования Оккам, предложенного двумя годами раньше. Основные сведения по транспьютеру и Оккаму приведены ниже.

Транспьютер и схожие с ним элементы, получившие название «транспьютероподобные элементы» (ТПЭ), весьма перспективны для построения средств поддержки интеллектуальных систем, а также для вычислительных средств следующих поколений. Различные фирмы, такие, как Intel и Motorola, конкурируют

в изготовлении новых моделей ТПЭ. Они объявили о разработке ТПЭ, более мощного, чем широко известный транспьютер T414 фирмы INMOS (64-разрядный вычислительный модуль с большим объемом ОЗУ и с расширенной системой команд). Намечена разработка ТПЭ с повышенной скоростью обмена информацией и с увеличенным числом портов (в пределе до 9). Предусмотрена разработка ТПЭ с сокращенным набором команд. Таким образом, в общем случае под ТПЭ подразумевается вычислительный модуль со следующими данными:

процессор (16-, 32- или 64-разрядный) с расширенной системой команд или, напротив, с сокращенным набором команд (обзор основных архитектурных решений по таким процессорам дан ниже);

оперативное запоминающее устройство емкостью 2 Кбайт и более;

число портов 4, 6, 8;

наличие встроенного механизма синхронизации работы по всем направлениям.

Разработка подобных вычислительных модулей неразрывно связана с развитием и совершенствованием новых технологий изготовления микросхем, систем автоматизированного проектирования, методов и средств самодиагностики, тестирования и восстановления отказавших модулей, методов и средств построения самосинхронных схем (см. § 4.3), с созданием принципиально новых языковых средств описания процессов обработки информации в среде из мощных вычислительных модулей типа ТПЭ, с подготовкой специалистов в этой сравнительно новой области.

Основным применением будущих ТПЭ предполагаются высокопроизводительные вычислительные системы новых поколений и многочисленные рабочие станции. Сведений о создании коммерческих интеллектуальных систем на базе ТПЭ пока нет, хотя, по мнению многих специалистов, эта перспективная область будет в ближайшие годы активно развиваться. По мере проработки вопросов реализации алгоритмов обработки информации в интеллектуальных системах будут произведены уточнения функционального назначения соответствующих ТПЭ и их систем команд путем физического моделирования таких систем. При этом наиболее перспективным направлением создания элементной базы для интеллектуальных систем следует считать выпуск соответствующих модулей на базе заказных СБИС, изготавливаемых с применением развитых средств систем автоматизации проектирования по требованиям заказчика.

На выбор элементной базы третьего уровня существенное влияние оказывает учет требований, намечаемых заказчиком на проектируемые структуры (например, требования построения легко тестируемых схем или построение самосинхронных схем). Для проектирования микросхем со сверхбольшой степенью интеграции эти требования часто становятся определяющими.

Существуют и постоянно развиваются методы построения узлов дискретных схем, тестируемых лишь несколькими наборами (проверяющего теста) [Горяшко, 1982а]. Учет этих методов при проектировании микросхемы путем введения избыточности как в элементный базис, так и в саму схему может привести к существенному повышению основного показателя изготовления микросхем — процента выхода годных. Обзор известных результатов в области синтеза легкотестируемых схем приведен в [Горяшко, 1982б].

То же относится к построению самосинхронных схем. Методы синтеза узлов дискретных устройств, устойчивых к состязаниям, параметрическим отказам, и работающих по реальным задержкам элементов исследуются в рамках научного направления под названием «Апериодическая схемотехника» (см. § 4.3). Основные идеи этого направления состоят в следующем.

В синхронных схемах требования к элементам, ограничивающим тактовое питание, сводятся к созданию мощного источника синхронизирующего сигнала из-за необходимости послышки его ко многим потребителям. Дополнительные усилители вносят задержку и тем самым снижают скорость работы схем. Кроме того, скорость работы зависит от частоты тактового сигнала, который не может повышаться беспредельно. В асинхронных схемах нет синхронизации и, следовательно, связанных с нею недостатков. Однако при построении асинхронных схем

требуются специальные схемные средства борьбы с состязаниями, что в ряде случаев также приводит к необходимости введения задержек. В самосинхронных схемах отсутствуют все перечисленные недостатки. Проектируемые устройства работают с максимальной скоростью, определяемой реальными задержками, но необходимость использования специальных сигналов, фиксирующих момент окончания переходных процессов в элементах, требует специальных аппаратных затрат.

При переходе от синхронных схем к самосинхронным (аперiodическим) необходимо примерно двукратное увеличение оборудования. Это в основном относится к схемам из функциональных элементов И-НЕ, ИЛИ-НЕ, И-ИЛИ-НЕ, сложность которых определяется по Шеннону, т. е. суммарным числом входов и выходов элементов схемы (уровень вентилей). Ряд схем (например, разного типа счетчики) требуют значительно меньшего дополнительного расхода оборудования при переходе к самосинхронной реализации. Переход с уровня вентилей на уровень транзисторов дает более оптимистические оценки. В частности, для КМОП-технологии, наиболее широко используемой в СБИС, переход к самосинхронной реализации требует не более 20—30 % увеличения оборудования. Это связано с наличием у КМОП-схем элементов двусторонней проводимости, которая эффективно используется при синтезе аперiodических схем [Варшавский и др., 1976].

Успехи микроэлектронной технологии уже в ближайшем будущем позволят создавать компактные микросхемы сверхбольшой степени интеграции, размещающие до  $10^7$  активных элементов. Это создаст в одном кристалле чрезвычайно сложные устройства. Аппаратурные затраты становятся малосущественным фактором — гораздо важнее построение надежных микросхем с требуемыми свойствами.

### Транспьютер и язык Оккам

Транспьютер — это программируемый СБИС-прибор, представляющий собой микропроцессор фон-неймановского типа с развитыми средствами связи [Walker, 1985]. Оккам — язык программирования, предназначенный для реализации программ на одном или нескольких взаимосвязанных транспьютерах. Причем в обоих случаях используется одна и та же технология программирования.

Отличие транспьютера от однокристалльной микроЭВМ в основном состоит в простоте и удобстве связи с аналогичными микросхемами: выходы одного транспьютера непосредственно подсоединяются ко входам других по всем направлениям в соответствии с конфигурацией сети. На рис. 4.1 приведена схема соединения двух транспьютеров (в первом указаны основные функциональные блоки). Таким образом, под связью между двумя транспьютерами подразумевается наличие двух линий связи, каждая из которых предназначена для передачи данных лишь в одном направлении.

Наличие средств связи по нескольким направлениям позволяет рассматривать транспьютер как готовый прибор для включения его в сеть. Предполагается, что сеть может быть организована так, что каждый транспьютер в ней решает локальную задачу, используя собственную память и информацию, поступающую из сети (по общей шине или по связям с соседями). Обмен информацией между процессором, собственной памятью и средствами связи осуществляется через внутреннюю шину. Поэтому локальных шин в сети транспьютеров должно быть не меньше, чем транспьютеров.

Структурная схема транспьютера IMS T414 приведена на рис. 4.2. Система команд процессора ориентирована на выполнение уплотненных (сжатых) программ, на применение языков высокого уровня и на поддержку параллельности в обработке данных. Оперативная память — быстрая, статическая, т. е. не нуждается в непрерывной регенерации, пока включено питание.

Блок ИВЗУ представляет собой 32-разрядный мультиплексный интерфейс с временной реконфигурацией, поддерживающий все типы памяти и предоставляющий доступ к линейному 4-гигабитовому адресному пространству. Максимальная скорость передачи данных в режиме прямого доступа составляет



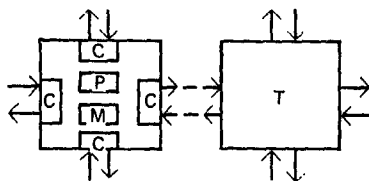


Рис. 4.1. Схема соединения двух транспьютеров:

Т — транспьютер; Р — процессор;  
М — память; С — средства связи

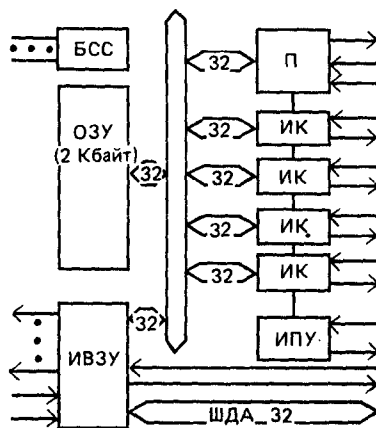


Рис. 4.2. Схема транспьютера  
IMS T414:

П — 32-разрядный процессор; БСС — блок системного сервиса; ОЗУ — внутреннее оперативное запоминающее устройство; ИВЗУ — интерфейс внешнего запоминающего устройства; ИК — интерфейс канала; ИПУ — интерфейс периферийных устройств; ШДА — шина данных и адреса

25 Мбайт/с. Четыре блока ИК (последовательной связи) обеспечивают одновремениую параллельную передачу сообщений к четырем другим транспьютерам со скоростью 10 Мбит/с по каждому из четырех дуплексных каналов [Ipotos, 1984]. В качестве средства синхронизации для небольших транспьютерных систем можно использовать внешний синхрогенератор с частотой 5 МГц, снабжающий сигналами собственные синхрогенераторы транспьютеров для формирования внутренних сигналов синхронизации работы всех узлов микросхемы. Микросхема транспьютера содержит около 250 000 транзисторов и построена с применением прогрессивной двухмикронной КМОП-технологии. Рассеиваемая мощность составляет не более 1 Вт. Керамический корпус содержит 84 вывода.

На базе транспьютеров путем их соединения с помощью имеющихся в них средств связи могут быть построены сложные вычислительные системы. Одни и те же программы могут быть перестроены для прогона на одном, десятках или тысячах транспьютеров с различными показателями «стоимость-выполнение». Если транспьютер используется в качестве единственного вычислителя, на нем может быть реализована программа с применением любой версии языка высокого уровня.

Язык Оккам создан для программирования процессов в сложных системах, состоящих из взаимосвязанных и взаимодействующих подсистем, допускающих параллельную работу. Каждый процесс в такой сложной системе представляется как «оккамовский», т. е. рассматривается как «черный ящик», работающий со своей локальной информацией. Процессы взаимодействуют посредством поименных информационных каналов. Совокупность взаимодействующих оккамовских процессов можно, в свою очередь, рассматривать как оккамовский процесс, что дает возможность структурного представления иерархии процессов.

Процессу можно поставить в соответствие любой объект, служащий целям переработки дискретной информации. Например, логический элемент, преобразователь кодов, часть микроЭВМ, микропроцессор, специализированный вычислитель и, наконец, специально созданный базовый элемент распределенной обработки информации — транспьютер. Процессы во времени конечны: каждый выполняет некоторое число действий и заканчивается. Действие, в свою очередь, может также рассматриваться как процесс или совокупность последовательных,

параллельных или альтернативных процессов (из последних выполняется лишь тот, которому соответствуют определенные выполненные условия).

Можно провести следующую параллель между языком Оккам и языком булевых функций: первый является таким же средством описания и проектирования транзисторных систем, как булева алгебра для описания и проектирования дискретных схем из логических элементов [May et al., 1984]. Конструирование вычислительной системы на транзисторах под решаемую задачу упрощается за счет отмеченных выше соответствий между языком Оккам и транзистором. Если язык Оккам используется для реализации программы на одном транзисторе, время выполнения программы распределяется между отдельными процессами с учетом возможности параллельного их выполнения (точнее, в псевдопараллельном режиме разделения времени), а коммутация соответствующих каналов (инициаторов и результатов процессов) осуществляется блоками данных, продвигающихся в пространстве памяти. Если же используется сеть транзисторов, каждый из них реализует отдельный процесс, а коммутация каналов выполняется на уровне связей между транзисторами.

В качестве примера прямой связи между языком Оккам и транзистором можно указать на имеющуюся в Оккаме процедуру пересылки данных из одного процесса в другой, что соответствует в аппаратной поддержке Оккама пересылке данных между транзисторами. В каждом транзисторе имеется возможность с помощью интерфейса связи переслать данные в другой транзистор в режиме прямого доступа к его памяти, освободив при этом собственный процессор для выполнения другой работы.

Программа на языке Оккам строится с помощью следующих трех примитивных процессов: входной процесс:  $c?v$  — передача переменной  $v$  текущего значения из канала  $c$ ; выходной процесс:  $c!e$  — передача текущего значения выражения  $e$  в канал  $c$ ; процесс присваивания:  $v := e$  — присваивание переменной  $v$  текущего значения выражения  $e$ .

Примитивные процессы комбинируются из конструкторов следующих трех типов [May, et al., 1984]:

SEQ (sequential) — процессы выполняются один за другим;

PAR (parallel) — процессы выполняются вместе;

ALT (alternative) — выполняется лишь первый, готовый к выполнению компонент процесса (процесс).

Конструкторы сами являются процессами и могут быть использованы как компоненты других конструкторов. Последовательной программе соответствует последовательность процессов, представляющих собой объединение примитивных процессов в последовательные конструкции. При этом могут еще использоваться обычные конструкторы (операторы) IF и WHILE. Сопотекающие процессы могут быть отражены в программе с помощью выражений, составленных из примитивных входных и выходных процессов, объединенных в параллельные или альтернативные конструкции.

Каждый канал в Оккаме является связующим звеном между двумя процессами. Все связи синхронизируемы и только тогда имеют место, когда оба (выходной и входной) процесса готовы к взаимодействию. И тогда данные выходного процесса копируются и передаются во входной процесс.

При разработке транзистора предпринимались усилия, направленные на облегчение программистского труда. Фирмой INMOS разработаны системы, позволяющие осуществить общее редактирование, компилирование и исправление ошибок на уровне исходных программ, ориентированных как на использование единственного транзистора, так и на системы из нескольких транзисторов с применением таких языков высокого уровня, как Си, Паскаль, Фортран и Оккам.

В последние годы создана новая версия — язык Оккам 2. Основные конструкторы языка остались без изменений. Получили развитие типы данных: они существенно расширены, появилась возможность их преобразования. Основное отличие языка Оккам 2 — это большее удобство для программиста из-за расширения

ных структур данных. В языке Оккам 2 возможно представление сложной структуры данных.

Транспьютер T414 — не единственный представитель семьи транспьютеров. Формой T4M33 выпускается также 16-разрядный транспьютер в корпусе с 64 выводами. Основные показатели этого прибора те же, что и у транспьютера T414. Некоторое отличие (кроме 16-разрядного процессора) в организации блока интерфейса памяти. Этот блок обеспечивает прямую адресацию 64-Кбайтного адресного пространства с максимальной скоростью передачи данных 20 Мбайт/с. Данные и адреса не мультиплексированы. Рассеиваемая мощность составляет 40 Вт.

Другие изделия той же фирмы IMS T800 — 32-разрядный транспьютер с плавающей точкой, IMS M212 — периферийный контроллер (peripheral controller).

Приборы связи IMS C004 — переключатель канала связи (communications link switch); IMS C011, IMS C012 — адаптеры канала связи (communications link adaptor).

Как уже отмечалось, семья транспьютероподобных элементов постоянно пополняется новыми изделиями, и многие фирмы включились в производство ТПЭ. В ряде стран предпринята попытка смоделировать транспьютер в виде платы с микросхемными микроэлементами предельной степени интеграции. Такие изделия известны под названием «псевдотранспьютер». Они перспективны как элементы систем моделирования на базе физических макетов процессов обработки дискретной информации и в том числе в интеллектуальных системах. Результаты моделирования могут быть использованы при разработке архитектур будущих микросхем ТПЭ. Однако о построении реальных вычислительных систем будущих поколений на основе псевдотранспьютеров не может быть и речи. В таких системах возрастет длина коммуникационных связей, что приведет к существенному снижению быстродействия из-за задержек в проводах. Возрастет также сложность решения задачи синхронизации процессов из-за разбросов задержек.

### Машины с сокращенным набором команд

В начале 80-х годов появился ряд зарубежных публикаций по машинам с сокращенными системами команд (RISC-машинам). Цель разработки таких машин состояла в увеличении эффективности выполнения программ на языках высокого уровня и в дальнейшем снижении показателя стоимость-производительность для вычислительных систем.

Такие же цели ставили разработчики других машин, однако общая тенденция развития архитектуры к началу 80-х годов состояла в ее усложнении, что в первую очередь было связано с расширением систем команд, их усложнением. Наиболее ярко это выразилось в переходе от PDP-11 к VAX, от IBM Система/3 к IBM Система/38, от IBM 360 к IBM 370, а также в появившихся машинах ориентированных на языки высокого уровня [Майерс, 1985].

Новизна разработки RISC-машин состояла в упрощении (а не в усложнении, как было принято ранее) архитектуры машин. Основы разработки RISC-машин заложены в [Patterson et al., 1980a]. Критика машин со сложной архитектурой приводится в более ранней работе [Patterson et al., 1980b]. Критический комментарий к работе [Patterson et al., 1980a] имеется [Clark et al., 1980]. Из существующих к началу 80-х годов машин к RISC-машинам наиболее близки машины С. Крея CDC 6600, Cray-1. Общеизвестно, что принципы построения этих машин положены в основу RISC-машин.

Наиболее известными первыми проектами RISC-процессоров были RISC-1 [Patterson et al., 1981], MIPS [Hennessy et al., 1981] и IBM 801 [Radin, 1982]. Первые два проекта велись соответственно в университете Беркли и Стэнфордском университете, а третий проект — в Ватсоновском исследовательском центре фирмы IBM. Известно также [Patterson et al., 1980a], что исследования велись в Белловских лабораториях.

Первыми исследования по RISC-машинам в октябре 1975 г. начали в фирме IBM, их инициатором был Дж. Коук, ему впоследствии была присуждена премия имени Тьюринга Ассоциации по вычислительной технике за 1987 г. за изобретение RISC-технологии [Электроника, 1987б]. В этих исследованиях участвовало около 20 человек, которые одновременно разрабатывали аппаратуру, управляющую программой и оптимизирующий компилятор [Rafin, 1982, 1983]. Исследования базировались на трех основных принципах

1 В систему команд включались наиболее часто используемые команды, которые можно реализовать за один достаточно примитивный такт машины с аппаратным управлением («минимизация такта»).

2 Предполагалось построить иерархическую память процессора и реализовать команды безусловного и условного перехода так, чтобы минимизировать количество тактов, когда процессор простаивает из-за блокировок («минимизация тактов ожидания») или «завершать выполнение команды на каждом такте».

3 Предполагалось построить компилятор с высокой степенью оптимизации программ, который бы производил максимальное количество действий, предписанных в программе, на этапе компиляции и генерировал бы наиболее оптимальные последовательности простых команд машины («минимизация тактов выполнения за счет эффективной компиляции»).

Первый и третий принципы требуют дополнительных пояснений.

Первый принцип — «минимизация такта». В машинах 70-х годов в системах команд содержались команды различной степени сложности, например загрузка, запись, сравнение, условный переход, сложение и пересылка байтов, перекодировка, операции над числами с плавающей точкой. Эффективная реализация сложных команд с совместным использованием аппаратного и микропрограммного управления обычно приводила к тому, что такт машины становился несколько большим по сравнению с тем, что нужно было для реализации простых команд, да и стандартизация реализации управления выполнением команд обычно приводила к увеличению количества тактов, за которые реализуются простые команды. Такое явление получило название «правило 4-1 команд». Его можно сформулировать так, если в процессе с тактом  $T$  есть  $n$  реализованных команд со средним временем выполнения  $t$ , то реализация в процессоре новых команд может увеличивать такт  $T$  и/или среднее время выполнения  $t$  для имеющихся  $n$  команд. Ясно, что такое правило — не физический закон, а скорее предостережение, обобщение опыта конструирования.

Другое явление, которое также было выявлено на машинах 70-х годов — это то, что подавляющая часть времени выполнения программы обычно приходится на выполнение небольшого числа достаточно простых команд. Для примера в табл. 4.1 приведены данные о статистическом и динамическом использовании команд IBM 360 на некотором наборе задач, имеющихся в [Майерс, 1985]. Хотя в табл. 4.1 приводятся данные по частоте использования, они в основном соответствуют данным по времени, затрачиваемым на выполнение команд.

На разных классах задач набор наиболее часто используемых команд может отличаться, однако общее правило таково, что количество команд в наборе невелико. Было даже сформулировано «правило 80/20»: 80% времени выполнения любой программы обычно приходится на выполнение команд, составляющих не более 20% всего набора команд машины.

Таким образом, получалось, что машины усложнялись, но это приводило к напрасным затратам времени при выполнении реальных программ. Эту ситуацию и пытались исправить, выдвигая в качестве основных первый принцип «минимизация такта».

Третий принцип — разработка компилятора с высокой оптимизацией, может, несколько противоречил общим тенденциям того времени, поскольку в основном за счет усложнения архитектуры машина стремилась к снижению сложности программного обеспечения и, в частности, компилятора.

Исследования в Ватсонском центре фирмы IBM по проекту 801 завершились в 1979 г. разработкой опытного образца машины IBM 801, которая работала почти в два раза быстрее, чем IBM 3033. Результаты были опубликованы

**Характеристики наиболее часто используемых команд IBM 360  
на некотором наборе задач**

Команда	Статистическая частота появления, %	Команда	Динамическая частота появления, %
L	28,6	L	27,3
TS	15,0	BC	13,7
BC	10,0	ST	9,8
LA	7,0	C	6,2
SR	5,8	LA	6,1
BAL	5,3	SR	4,5
SLL	3,6	IC	4,1
IC	3,2	A	3,7

Примечание. L — загрузка из памяти в регистр 32-разрядного целого; ST — запись в память из регистра 32-разрядного целого; BC — переход по условию; LA — загрузка адреса в регистр; BAL — переход с возвратом; SLL — сдвиг влево логический, IC — прочитать символ; SR — вычитание целых; C — сравнение 32-разрядных целых; A — сложение целых.

лишь в 1982 г. [Radin, 1982]. Как выяснилось, в 1977 г., в другом исследовательском центре этой фирмы, начались работы по другому проекту, который использовал в значительной степени результаты проекта 801, — проект ROMP (Research-OPD-MicroProcessor, OPD-IBM Office Product Division в Аустине). Фирмой IBM были истрачены сотни миллионов долларов [Barney, 1985] на создание микропроцессора ROMP, базирующегося на принципах, отработанных в проекте IBM 801 [Hopkins, 1987; Simpson, 1986; Simpson et al., 1987]. Одно из основных усовершенствований в ROMP по сравнению с IBM 801 состояло в достижении хорошей экономии памяти за счет более компактного представления программ. Микропроцессор ROMP является основой инженерной рабочей станции фирмы IBM PC/RT [Nguyen, 1987]. Он также используется как базовый процессор вычислительного узла в супермашине фирмы IBM RP3. Эта машина должна содержать до 512 вычислительных узлов и иметь пиковую производительность до 1,4 млрд. опер./с.

Проекты RISC-I и MIPS тоже были успешно продолжены и имели внедрение в промышленности. [Patterson et al., 1982, 1983, 1984; Katevenis et al., 1984; Unger et al., 1984; Taylor et al., 1986; Patterson, 1987] — наиболее известные из работ группы университета Беркли; реализация микропроцессора MIPS в Стэнфордском университете описывается в работах [Hennessy et al., 1982a; Przybylski et al., 1984; Gross, 1985]; усовершенствованный микропроцессор MIPS-X Стэнфордского университета — в [Chow et al., 1987; Steenkiste et al., 1987]; промышленный вариант микропроцессора MIPS микропроцессор R200 фирмы MIPS Computer Systems Inc — в [Ohr, 1986; Moussouris et al., 1986; MIPS, 1987; Chow et al., 1987]; использование архитектуры MIPS при разработке микропроцессоров на арсениде галлия (GaAs), работающих с тактовой частотой 200 МГц, описывается в [Milutinovic et al., 1986; Computer, 1986; Naused et al., 1987].

Работы по RISC-машинам в Белловских лабораториях также завершились разработкой микропроцессора CRISP, но он появился несколько позже и уже использовал отдельные положительные элементы более сложных машин, которые не снижают достоинств, достигаемых в RISC-архитектуре, а дополняют ее [Ditzel et al., 1987].

Проведенные в рамках проектов IBM 801, RISC-I, RISC-II и MIPS исследования позволили сделать определенные обобщения [Hennessy, 1984; Patterson, 1985]. Они также в значительной степени обогатили опыт разработки программных систем. Компиляторы и операционные системы для RISC-машин описыва-

ются в работах [Auslander et al., 1982; Chaitin, 1982; Hennessy et al., 1983; Chow et al., 1986; DeMoney et al., 1986; Milutinovic et al., 1987].

В 1982—1985 гг. дискуссии по RISC- и CISC-машинам (Complex Instruction Set Computers — машины со сложной системой команд) были наиболее острыми. Одновременно всё больше промышленных фирм ориентировалось на этот класс машин, однако практики, используя в своих изделиях лучшие решения RISC- и CISC-архитектур, отдавали предпочтение RISC.

Среди промышленных фирм в начале 80-х годов наиболее активными сторонниками RISC-машин были фирмы Pyramid, Ridge, Hewlett-Packard, которые вели разработки в классе супермини-машин. В классе супермашин элементы RISC-машин были использованы фирмой Nippon Electric Computer Corp. (NEC, Япония) при разработке скалярного процессора машин серии SX.

В настоящее время RISC-архитектура наиболее популярна [Манузов, 1987а; Майерс и др., 1986; Электроника, 1987а], однако CISC-архитектура, во всяком случае ее элементы, используется в подавляющем большинстве выпускаемых промышленностью современных RISC-машин [Gimarc et al., 1987; Mokhoff, 1986], выпускаются также CISC-машинны и имеются их принципиальные сторонники [Flynn et al., 1987; Смит, 1987; Уоллер, 1987; Мануэль, 1987], которые негативно относятся к RISC-машинам, но их не так много.

Так или иначе, изобретение и развитие RISC-технологии заставило вновь анализировать системы команд, искать новые архитектурные решения. В качестве основных этапов разработки RISC-машин выдвигаются следующие [Gimarc et al., 1987]:

- анализ области использования для определения наиболее часто применяемых операций и выделение соответствующих команд;

- оптимизация структуры разрабатываемого процессора с целью наиболее быстрого выполнения этих выделенных команд;

- введение других команд, если это не усложняет процессор (они также часто выполняются и их добавление не замедляет выполнения ранее введенных команд);

- применение такого же подхода при разработке других элементов вычислительной системы;

- перенос как можно большей части действий из аппаратуры в программное обеспечение (компилятор).

Процессоры, разработанные с использованием такого подхода, называются RISC-процессорами.

Обычно выделяют следующие основные свойства RISC-машин, хотя они не являются необходимыми:

- выполнение большей части команд за один такт;

- система команд имеет тип «загрузка-запись», когда обрабатываемые данные выбираются из регистров и записываются только в регистры, а обращение к памяти возможно лишь в командах загрузки в регистр, записи из регистра в память, а также в командах перехода, обращения к подпрограммам;

- дешифрация команд производится аппаратно;

- в системе команд относительно мало операций и мало различных режимов адресации операндов;

- формат команд достаточно простой для дешифрации и выборки;

- сложность реализации функций, необходимых пользователю, переносится, если это возможно, в компилятор;

- высокая степень конвейеризации выполнения команд;

- большое количество регистров (с окнами или без них);

- в процессоре имеется много уровней иерархии памяти;

- система команд чаще всего разрабатывается для специальных областей использования.

**Процессор RISC-II.** Обрабатывает двоичные коды и числа с фиксированной точкой в 8, 16 и 32 разряда, длина адреса 32 разряда, имеет 32 адресуемых регистра общего назначения, причем регистр 0 всегда содержит нуль (рис. 4.3).



Рис. 4.3. Формат команд процессора RISC-II:  
OPCODE — поле кода операции

Имеется четыре типа команд: регистр-регистр, регистр-память, команды перехода, специальные команды. Первые три формата — основные (далее рассматриваются подробно).

Команды типа регистр-регистр — трехадресные. Для этих команд:

DEST — номер регистра, в который помещается результат операции;

SOURCE1 — номер регистра с первым операндом;

SOURCE2 — номер регистра со вторым операндом или 13-разрядная константа в зависимости от значения IMM (если  $IMM=0$ , то это номер регистра, если  $IMM=1$ , то это 13-разрядная константа, которая перед выполнением операции расширяется до 32-разрядного слова умножением знакового разряда влево).

Команды этого типа могут вырабатывать или не вырабатывать признак результата в зависимости от содержимого поля SCC (если  $SCC=0$ , то код признака результата операции не устанавливается, если  $SCC=1$ , то устанавливается). Обрабатываемыми данными команд типа регистр-регистр могут быть 32-разрядные числа с фиксированной точкой и двоичные коды.

Команды типа регистр-память двухадресные. Они могут быть лишь команды загрузки в регистр и записи из регистра в память. Используется 32-разрядный адрес, адресация до байта. Операндом в памяти может быть байт, 16-разрядное полуслово, 32-разрядное слово. Использование полей в командах типа регистр-память следующее:

DEST — регистр, в который при считывании помещается операнд из памяти либо из которого при записи в память берется записываемая информация;

SOURCE1 — индекс-регистр адреса-операнда из памяти;

SOURCE2 — 13-разрядное смещение.

Фактически допускается лишь способ адресации с индексацией, причем нет возможности задать одновременно базу и индекс, как в системе команд IBM 360/370. Необходимость использования таких команд встречается редко. Исключение такого способа адресации позволяет сэкономить аппаратуру, поскольку отпадает необходимость в использовании трехходового сумматора для вычисления адреса операнда. Используя то, что регистр 0 всегда хранит ноль, можно работать с операндами по абсолютным адресам. Для этого в качестве индекса-регистра нужно задать регистр 0. Команды рассмотренного типа устанавливают признак результата, если  $SCC=1$ .

К командам перехода относятся команды условной и безусловной передачи управления, команды обращения к подпрограммам и команда выхода из подпрограммы. В командах этого типа адрес, по которому передается управление, либо адрес точки входа в подпрограмму задается одним из следующих двух способов:

как адрес в командах типа регистр-память (индекс-регистр и смещение);

относительно текущего счетчика адреса (в этом случае 19-разрядное смещение относительно счетчика берется из полей команды SOURCE1, IMM и SOURCE2).

В командах условного перехода DEST задает условие перехода.

Система команд RISC-II представлена в табл. 4.2 (всего 39 команд, в таблице отсутствуют лишь несколько специальных команд). Ниже приводится расшифровка мнемонических обозначений кодов операций из табл. 4.2 и обозначений операндов:

Система команд процессора RISC-II [Sherburne et al., 1984]

Операция	Операнды	Комментарий
ADD	Rd, Rs, S2	$Rd := Rs + S2$
ADDC	Rd, Rs, S2	$Rd := Rs + S2 + \text{carry}$
SUB	Rd, Rs, S2	$Rd := Rs - S2$
SUBC	Rd, Rs, S2	$Rd := Rs - S2 - \text{borrow}$
SUBI	Rd, Rs, S2	$Rd := S2 - Rs$
SUBCI	Rd, Rs, S2	$Rd := S2 - Rs - \text{borrow}$
AND	Rd, Rs, S2	$Rd := Rs \& S2$
OR	Rd, Rs, S2	$Rd := Rs \vee S2$
XOR	Rd, Rs, S2	$Rd := Rs \oplus S2$
SLL	Rd, Rs, S2	$Rd := Rs \ll S2$
SRL	Rd, Rs, S2	$Rd := Rs \gg S2$
SRA	Rd, Rs, S2	$Rd := Rs \ggg A \ S2$
LDW	Rd, (Rx) S2	$Rd := M[Rx + S2]$
LDHU	Rd, (Rx) S2	$Rd := M[Rx + S2] \text{ (align, zero-fill)}$
LDHS	Rd, (Rx) S2	$Rd := M[Rx + S2] \text{ (align, sign-ext)}$
LDBU	Rd, (Rx) S2	$Rd := M[Rx + S2] \text{ (align, zero-fill)}$
LDBS	Rd, (Rx) S2	$Rd := M[Rx + S2] \text{ (align, sign-ext)}$
STW	Rm, (Rx) S2	$M[Rx + S2] := Rm$
STH	Rm, (Rx) S2	$M[Rx + S2] := Rm \text{ (align)}$
STB	Rm, (Rx) S2	$M[Rx + S2] := Rm \text{ (align)}$
JMPX	COND, (Rx), S2	if COND then $PC := Rx + S2$
JMPR	COND, Y	if COND then $PC := PC + Y$
CALLX	Rd, (Rx) S2	$Rd := PC; PC := Rx + S2; CWP -$
CALLR	Rd, Y	$Rd := PC; PC := PC + Y; CWP -$
RET	(Rx) S2	$PC := Rx + S2; CWP ++$
LDHI	Rd, Y	$Rd < 31:13 > := Y; Rd < 12:0 > := \emptyset$
GTLP	Rd	$Rd := \text{last PC}$
GETPSW	Rd	$Rd := \text{PSW}$
PUTPSW	Rm	$\text{PSW} := Rm$
RETI	(Rx) S2	$PC := Rx + S2; CWP ++; \text{en.intr.}$
CALLI		on intr.: $CWP -$ ; $R25 := \text{last PC}$ ; $PC := \text{Intr. Vect.}; \text{disable intr.}$

ADD — сложение целых;  
 ADDC — сложение целых с учетом переноса;  
 SUBI — обратное вычитание целых;  
 SUBC — вычитание целых с учетом заема;  
 SUBI — обратное вычитание целых;  
 SUBCI — обратное вычитание целых с учетом заема;  
 AND — побитовое И;  
 OR — побитовое ИЛИ;  
 XOR — побитовое Исключающее ИЛИ;  
 SLL — логический сдвиг влево;  
 SRL — логический сдвиг вправо;  
 SRA — арифметический сдвиг вправо;  
 LDW — загрузка слова;



LDHU — загрузка полуслова без учета знака;  
 LDHS — загрузка полуслова с учетом знака;  
 LDBU — загрузка байта без учета знака;  
 LDBS — загрузка байта с учетом знака;  
 STW — запись слова;  
 STH — запись полуслова;  
 STB — запись байта;  
 JMPX — условный переход индексированный, отложенный;  
 JMPR — условный переход относительно счетчика адреса команд, отложенный;  
 CALLX — вызов подпрограммы по индексированному адресу с изменением регистрового окна;

CALLR — вызов подпрограммы по адресу относительно счетчика адреса команд с изменением регистрового окна;

RET — возврат из подпрограммы с восстановлением регистрового окна;

LDHT — загрузка задаваемой непосредственно в команде 19-разрядной константы в старшие разряды;

GTLPС — рестарт конвейера выполнения команд после прерывания;

GETPSW — чтение слова состояния программ;

PUTPSW — запись слова состояния программ;

RETI — выход из обработки прерывания по индексированному адресу с восстановлением регистрового окна и разрешением прерываний;

CALLI — это особая команда процессора, она реализована непосредственно в аппаратуре (пользователю недоступна), выполняется при возникновении прерывания: изменяется регистровое окно, в регистр 25 записывается значение счетчика команд, далее счетчик команд устанавливается по вектору и закрываются прерывания;

Rd, Rs, Rx, Rm — это номера 32-разрядных регистров общего назначения;

S2 — либо номер 32-разрядного регистра общего назначения, либо 13-разрядная константа;

M [Rx+S2] — это 32-разрядное слово, либо 16-разрядное полуслово, либо байт, расположенные в оперативной памяти по адресу Rx+S2;

COND — 4-битовое условие перехода;

Y — 19-разрядная константа, задаваемая непосредственно в команде;

PC — счетчик адреса команд;

CWP — указатель текущего регистрового окна в регистровом файле.

Все команды из табл. 4.2 могут устанавливать признак результата. Команды сложения и вычитания с учетом разряда переноса и заема могут использоваться для программной реализации операций умножения и деления, которых в процессоре RISC-II нет.

Команды перехода и вызова подпрограмм RISC-II имеют особенности, которые позволяют добиться высокой эффективности выполнения программ. Использование в RISC-II решения по их реализации применяют во многих других RISC-процессорах и даже появившихся CISC-процессорах. Рассмотрим эти решения более подробно.

Процессор RISC-II — конвейерный. Ступени конвейера изображены на рис. 4.4 [Sherburne et al., 1984]. Время такта (ступень конвейера) в наиболее отработанном варианте реализации RISC-II равно 440 нс ( $n$ -МОП трехмикронная технология).

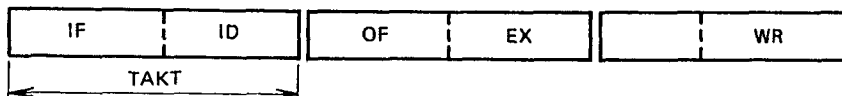


Рис. 4.4. Ступени конвейера выполнения команд процессора RISC-II:

IF — выборка команд; ID — дешифрация команд; OF — выборка операндов; EX — выполнение команд в АЛУ; WR — запись результата в регистр общего назначения

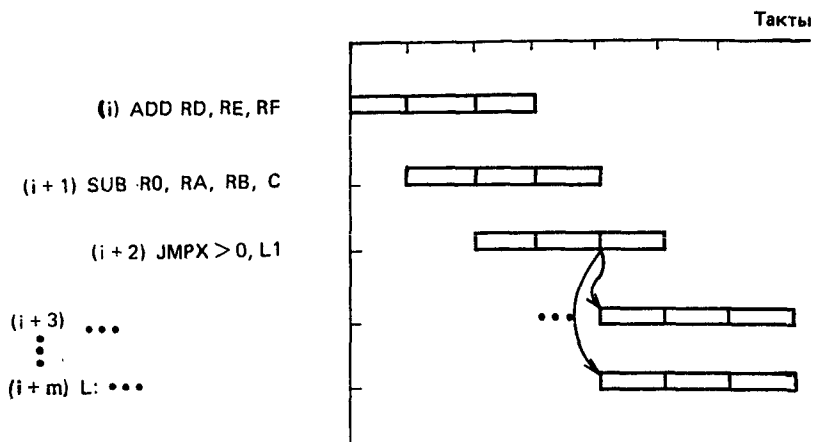


Рис. 4.5. Блокировка выполнения команды по адресу перехода  $(i+3)$  либо  $(i+m)$

Трудности выполнения команд условного и безусловного переходов в конвейерных процессорах хорошо известны. После выборки команды перехода некоторое время (пока не определится условие и/или, не вычислится адрес перехода) остается неясным, по какой ветви пойдет выполнение программы, т. е. адрес следующей для выполнения команды. Решения этой проблемы в больших машинах известны и до настоящего времени предлагаются все более эффективные варианты решений. Однако для микропроцессоров они слишком сложны.

В RISC-I и RISC-II, так же как и в MIPS, IBM 801, принято простое решение: считать команду, расположенную вслед за командой перехода, всегда выполняемой. Таким образом, выборка этой команды происходит в период, когда определяется условие перехода и вычисляется адрес перехода, а ее выполнение совмещено с выборкой команды по адресу перехода. Например, пусть имеются операторы языка высокого уровня

$$D = E + F$$

$$\text{IF (A.GT.C) GOTO L}$$

и все переменные находятся в регистрах, обозначим их RD, RE, RF, RA, RC. Будем для реализации этих операторов использовать команды ADD, SUB (с установкой признака условия) и JMPX. Если JMPX не была бы командой с отложенным переходом, то приведенные операторы были бы закодированы так, как показано на рис. 4.5. На этом же рисунке показано их выполнение в трехступенчатом конвейерном процессоре. Видно, что следующая после JMPX команда (по адресу  $i+3$  или L) будет выполняться с задержкой в 1 такт.

Поскольку в RISC-II условные и безусловные переходы отложенные, то приведенные операторы следует программировать так, как указано на рис. 4.6. Из этого рисунка видно, что в данном случае блокировки нет.

Предложенное в RISC-II решение по организации переходов приемлемо, поскольку в основе RISC-технологии лежит ориентация на использование языков высокого уровня и эффективных компиляторов.

Следует отметить, что введение отложенных переходов не снимает полностью проблемы эффективной организации передач управления в конвейерных процессорах, хотя и является прогрессивным. Более того, оно обуславливает возникновение некоторых трудностей при реализации в случае многоступенчатых конвейеров, а также при организации действий после возникновения прерываний. Работы по этой проблеме продолжаются, опубликованы новые интересные решения [McFarling, Hennessy, 1986; Ditzel, McLellan, 1987].

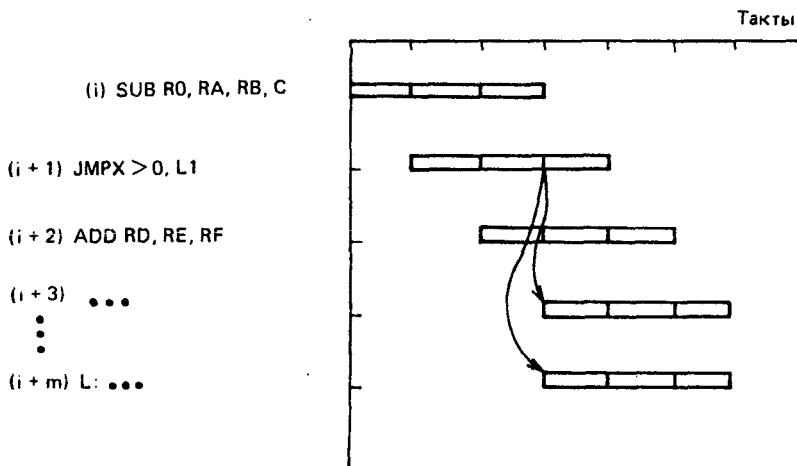


Рис. 4.6. Выполнение команды с отложенным переходом, блокировки команды (i+3) или (i+m) нет

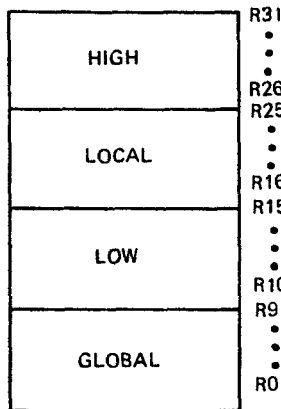
Суть эффективной поддержки обращений к подпрограммам в RISC-II состоит в следующем.

1. Предполагается не запоминать значения регистров в точке обращения к подпрограмме и соответственно не восстанавливать их значения при возврате, а просто физически подменять программно-доступные регистры новыми регистрами из некоторого файла регистров.

2. Для повышения эффективности передачи параметров физические области, на которые отображаются программно-доступные регистры, предлагается сделать перекрывающимися.

3. Поскольку в программах всегда имеются глобальные переменные, доступные из различных подпрограмм, то часть программно-доступных регистров предлагается всегда отображать на одной и той же физической области регистрового файла.

Более конкретно в RISC-II эти принципы реализуются так. Программно-доступные регистры разделены на подобласти, как указано на рис. 4.7. Всего



имеется четыре подобласти: HIGH — используется для передачи параметров из вызывающей подпрограммы; LOCAL — содержит локальные переменные подпрограммы; LOW — используется для передачи параметров в вызываемую подпрограмму; GLOBAL — глобальные переменные. Подобласти HIGH, LOCAL, LOW образуют регистровое окно.

При отображении программно-доступных регистров на регистровый файл области LOW вызывающей программы и HIGH вызываемой подпрограммы отображаются на одну и ту же область регистрового файла. Регистры области GLOBAL всегда отображаются на регистры файла с номерами R0—R9. Регистровый файл RISC-II содержит 138<sub>10</sub> регистров. Размещение такого большого количества регистров

Рис. 4.7. Подобласти программно-доступных регистров

**Система команд процессора MIPS на уровне языка Ассемблера**  
**[Przybylski et al., 1984]**

Операция	Сперанды	Комментарий
ADD	src1, src2, dst	dst := src2 + src1
AND	src1, src2, dst	dst := src2 & src1
IC	src1, src2, dst	dst[src1] := src2
OR	src1, src2, dst	dst := src2   src1
RLC	src1, src2, src3, dst	dst := src2    src3 <<< src1
ROL	src1, src2, dst	dst := src2 <<< src1
SLL	src1, src2, dst	dst := src2 << src1
SRL	src1, src2, dst	dst := src2 >> src1
SRA	src1, src2, dst	dst := src2 << A src1
SUB	src1, src2, dst	dst := src2 - src1
SUBR	src1, src2, dst	dst := src1 - src2
AC	src1, src2, dst	dst := src2 [src1]
XOR	src1, src2, dst	dst := src2 XOR src1
LD	A [src], dst	dst := M[A + src]
LD	[src1 + src2], dst	dst := M[src1 + src2]
LD	[src1 << src2], dst	dst := M[src1 << src2]
LD	A, dst	dst := M[A]
LD	I, dst	dst := I
MOV	src, dst	dst := src
ST	src1, A [src]	M[A + src] := src1
ST	src1, [src2 + src3]	M[src2 + src3] := src1
ST	src1, [src2 << src3]	M[src2 << src3] := src1
ST	src, A	M[A] := src
BRA	dst	PC := dst + PC
BRA	COND, src1, src2, dst	PC := dst; PCif COND(src1, src2)
JMP	dst	PC := dst
JMP	A[src]	PC := A + src
JMP	O A[src]	PC := M[A + src]
TRAP	COND, src1, src2	PC := ∅ if COND(src1, src2)
Save PC	A	M[A] := PC - 3
SET	COND, src, dst	dst := -1 if COND(src, dst) dst := ∅ if COND(src, dst)

внутри кристалла стало возможным благодаря тому, что в процессоре RISC-II простое устройство управления.

**Процессор MIPS.** Основная особенность процессора MIPS (Microprocessor without Interlocked Pipe Stages — микропроцессор без блокирующих друг друга ступеней конвейера) в его названии. В больших машинах работа той или иной ступени конвейера может быть задержана, если совместное выполнение соседних команд приводит к неверному результату, т. е. к результату, который не совпадает с результатом строго последовательного выполнения этих команд. В MIPS такие блокировки с целью экономии аппаратуры и сокращения такта процессора не предусмотрены. Разработчики MIPS оценивают, что такое решение позволило им сократить время такта процессора на 10%. Забота об обеспечении правильного выполнения команд в конвейере возлагается на компилятор или программу ассемблирования.

В микропроцессоре MIPS ассемблерное представление программ не перекодировается непосредственно в машинное представление. При переводе в машин-

ные коды отдельные команды могут быть переставлены для обеспечения правильности их выполнения в конвейере, соседние команды могут быть упакованы в одно слово, так как на уровне машинного представления в MIPS используется плотное кодирование команд. Ниже приводится описание команд MIPS в том виде, как их видит программист на языке ассемблера.

Процессор MIPS обрабатывает двоичные коды и числа с фиксированной точкой в 32 разряда. Длина адреса — 32 разряда. Имеется 16 регистров общего назначения, нет команд обращения к подпрограммам, зато есть команды, реализующие элементарные шаги операций умножения и деления (умножение на 2 разряда и анализ одного разряда при делении), есть поддержка преобразования виртуальных адресов в физические, а также есть команды, которых нет в RISC-II. Процессор MIPS содержит почти в два раза меньше транзисторов, чем RISC-II (25 тыс. против 45 тыс.).

Система команд процессора MIPS на уровне языка ассемблера приведена в табл. 4.3. Здесь отсутствует несколько команд, например шаг умножения и деления. Ниже приводится расшифровка mnemonicских обозначений кодов операций из табл. 4.3 и обозначений операндов:

ADD — сложение целых;  
AND — побитовое логическое И;  
IC — вставить байт из младших разрядов src2 в байт src1 регистра dst;  
RLC — конкатенация двух регистров и затем циклический сдвиг этой структуры на src1 позиций;  
SLL — логический сдвиг влево;  
SRL — арифметический сдвиг влево;  
SRA — арифметический сдвиг вправо;  
SUB — вычитание целых;  
SUBR — обратное вычитание целых;  
XC — выборка байта src1 из регистра src2 и помещение его в регистр dst;  
LD — несколько команд загрузки 32-разрядного слова в регистр, в зависимости от операндов различают: A [src], dst — загрузка слова с базированным адресом; [src1+src2], dst — загрузка слова с базированным индексированным адресом; [src1<<src2], dst — загрузка слова с предварительно сдвинутым базированным адресом; A, dst — загрузка слова по абсолютному адресу; I, dst — загрузка константы, заданной непосредственно в команде;

MOV — пересылка из одного регистра в другой;

ST — несколько команд записи 32-разрядного слова в память, в зависимости от операндов различают запись по базированному адресу, по базированному индексированному адресу, по абсолютному адресу;

BRA — безусловный переход относительно счетчика адреса, условный переход относительно счетчика адреса;

JMP — безусловный переход по непосредственному адресу, по базированному адресу, по косвенному адресу;

TRAP — установка прерывания, если выполняется заданное условие.

Save PC — запоминание счетчиков адреса команд для всех команд, находящихся в ступенях конвейера процессора. Команда применяется после возникновения прерывания и после срабатывания команды TRAP;

SET — установка значения регистра в зависимости от выполнения заданного условия.

dst, src1, src2, src3, src — номера регистров общего назначения, src1 и src3 могут быть малыми константами;

PC — счетчик адреса команды;

COND — условие, задаваемое в команде.

Конвейер обработки команд процессора MIPS сложнее, чем в RISC-II, его изображение приводится на рис. 4.8. Выборка команды производится каждые два такта. Отдельная команда ассемблерного уровня может не использовать все ступени конвейера, поэтому в процессоре MIPS на уровне машинных команд две различные команды могут быть упакованы в одну. Например, команда за-

IF	ID	OD	SX	OF
----	----	----	----	----

Рис. 4.8. Ступени конвейера MIPS:

IF — выборка команды, ID — дешифрация команды; OD — декодирование операнда; SX — запись результата или выполнение; OF — чтение операнда для команд загрузки

грузки в один регистр и увеличение другого регистра на единицу — это одна синтетизированная машинная команда.

Производительность RISC-II и MIPS оценивалась по времени прогона оценочных программ (бенчмарках). Она соответствует 2—3 MIPS (1 MIPS — производительность VAX 11/780).

### Современные RISC-процессоры

Имеется много RISC-процессоров, выпускаемых промышленностью, а также экспериментальных. Есть процессоры с элементами RISC и CISC-машин [Гош, 1987]. Рассмотрение ряда RISC-процессоров середины 80-х годов приводится в [Gimac et al., 1987]. В целом налицо тенденция усложнения RISC-процессоров, хотя есть и простые процессоры, причем это характерно для архитектур, ориентированных на реализацию с использованием высокочастотных схем (ECL, GaAs).

Всего в работе [Gimac, Milutinovic, 1987] рассмотрен 21 процессор различной ориентации (символьные преобразования, научные вычисления, коммерческие приложения, обработка сигналов, универсальные процессоры). Приводятся их количества команд, регистров, времена тактов, производительности, форматы команд, количество исполнительных блоков.

### 4.3 Аperiodическая схемотехника

*В. И. Варшавский, В. Б. Мараховский,  
Л. Я. Розенблюм, А. В. Яковлев*

Перспективным направлением исследований при создании аппаратной поддержки интеллектуальных систем является разработка новых динамических вычислительных моделей и архитектур с высшим уровнем децентрализации и параллелизма. Необходимость эффективной реализации парадигмы параллелизма потребовала ревизии сложившихся схемотехнических представлений. Одним из таких направлений служит концепция самосинхронизации, средства реализации которой поставляют *самосинхронная (аperiodическая) схемотехника* [Варшавский, 1976, 1986].

#### Принцип самосинхронности

Термин «самосинхронная» «аperiodическая» схемотехника подчеркивает отличие от двух традиционных схемотехнических подходов — синхронного и асинхронного. Первый, являющийся основным в вычислительной технике, связан с использованием механизма тактирования, задаваемого синхронизирующим устройством (таймерами, часами) и применяемого для согласования работы устройств во времени. Второй подход нашел применение лишь в тех системах дискретной автоматики, где синхронизация не используется. Синхронные схемы имеют ряд преимуществ, главным из которых является относительная простота по сравнению с асинхронными, в которых отсутствие тактового генератора связано с необходимостью введения избыточности для борьбы с аномалиями их

динамического поведения (состязаниями, риском). Использование асинхронного подхода не дает увеличения быстродействия, поскольку отсутствие часов трансформируется в требование соблюдения временных интервалов между подачей соседних входных воздействий.

Альтернативой традиционным подходам является самосинхронная схемотехника. При реализации самосинхронного подхода часы в схеме заменяются механизмом индикации, который не отсчитывает абсолютное время, а фиксирует те моменты времени, когда в схеме закончились переходные процессы, вызванные изменением входного воздействия. Моменты выдачи сигналов индикации определяются величинами реальных задержек, которые зависят от условий функционирования элементов схемы и могут варьироваться от нуля до любой конечной величины, что делает самосинхронные схемы устойчивыми к временной нестабильности элементов.

В синхронных и асинхронных схемах величина такта должна выбираться исходя из максимальных величин задержек элементов, и в случае превышения расчетного такта в реальной схеме правильность функционирования схемы разрушается. Таким образом, традиционные концепции становятся бессмысленными, если предельные значения величины задержек априори неизвестны. Для самосинхронных схем знание величин разбросов в жизненном цикле схемы не требуется: схема гарантированно правильно функционирует инвариантно к временным параметрам элементов. При этом требование физического наличия индикатора в самосинхронной схеме не обязательно. Роль индикатора могут выполнять специальные самосинхронные коды, наличие или отсутствие которых на выходе схемы несет для соединенных с ней устройств информацию о завершении в ней переходных процессов. Самосинхронные схемы напоминают синхронные тем, что они могут иметь выделенную в явном виде пару управляющих сигналов — запрос (аналог тактового сигнала, который выдается после установки информационных входов) и ответ, который квитирует получение запроса. При таком представлении переходный процесс в схеме относительно пары управляющих сигналов моделируется элементом задержки.

Самосинхронные схемы имеют преимущества и недостатки по сравнению с традиционными схемами; эти их особенности будут рассмотрены ниже.

В литературе встречаются следующие синонимы понятия «самосинхронные схемы»: самосинхронные (self-timed) схемы; схемы, не зависящие от скорости\* (speed-independent circuits); схемы, не зависящие от задержек элементов\*, схемы, нечувствительные к задержкам (delay-insensitive circuits); аперiodические\* (dead-beat) схемы; полумодулярные\* (semimodular) схемы; схемы Маллера\* (Muller's circuits). В толкованиях этих базовых понятий имеются некоторые разночтения. Те из них, которые помечены звездочками, чаще используются для наименования тех классов «чистых» схем, которые правильно функционируют в предположении о любых конечных величинах задержек элементов и их вариаций.

Основная задача аперiodической схемотехники формулируется так. Задается спецификация, моделирующая в определенном смысле некоторый реальный процесс. Эта спецификация анализируется с целью выявления некоторых как полезных, так и аномальных свойств процесса. Затем спецификация модифицируется с целью парирования аномалий, и по новому представлению (путем использования формальных или эвристических процедур) конструируется (синтезируется) схема, поведение которой совпадает с требуемой спецификацией. Для получения нового качества схем требуются как отличные от традиционных модели (динамического типа с возможностью отражения параллелизма), так и альтернативные методы анализа и синтеза.

### Краткая история

Теоретической базой схемотехники, как и вычислительной техники, является теория автоматов, где в качестве структурной модели асинхронного автомата господствующее место занимает модель Хаффмена [Huffman, 1954; Варшавский,

1976]. Двумя годами позднее Д. Е. Маллер, американский специалист по теории автоматов из Иллинойского университета, вместе со своим коллегой У. С. Бартки опубликовал отчет, в котором впервые было намечено новое направление, связанное с разработкой схем, поведение которых не зависит от скорости элементов. Эта основополагающая работа, к сожалению, почти не встретила ответной реакции специалистов. Много позже Р. Е. Миллер [Миллер, 1971] пытался обратить внимание читателей своей книги на маллеровский подход, но успеха не достиг, так как чрезмерное увлечение теоретическими аспектами модели оттолкнуло практиков, не увидевших никаких изящных схемных решений. Самого Маллера между тем волновали не только фундаментальные вопросы: он пытался воплотить свои мысли в рамках создания вычислительной системы *Illiac II*, для чего потребовалась разработка программного обеспечения анализа асинхронных схем на полумодулярность (что равносильно проверке их на принадлежность классу аperiodических) и соответствующей элементной базы. Однако эта попытка окончилась неудачей из-за недостаточного уровня технологической базы того времени (навесные элементы), слабой проработки схемотехники базовых узлов компьютера. Малая эффективность предложенных в то время схемных решений свела на нет ожидаемое увеличение быстродействия самосинхронных схем, вытекающее из организации их работы по реальным задержкам элементов. Правда, некоторые теоретики [Ангер, 1977; Фридман и др., 1978] обратили внимание на специфический запрос-ответный режим функционирования схем Маллера.

В 60-е годы наблюдалось становление новой научной дисциплины — теории сетей Петри, получившей название по имени немецкого ученого Карла Адама Петри, предложившего в 1962 г. новую модель информационных потоков в системах. Выдвинутая им идея была развита в США и других странах [Питерсон, 1984; Котов, 1984; Розенблюм, 1983]. Такие достоинства новой модели, как возможность отражения асинхронности и параллелизма, недетерминированности процессов, динамики их функционирования, простой синтаксис и наглядность модели в сочетании с достаточно широкими функциональными возможностями, явились причиной ее популярности. Теория сетей Петри развивалась самостоятельно, но тем не менее она может рассматриваться не только как автономная область науки, но и как ветвь теории автоматов в ее широком понимании, занимающая ранее не исследованную нишу между конечными автоматами и машинами Тьюринга. Подавляющая часть работ по сетям Петри «обслуживает» проблематику общей теории систем и параллельного программирования, но часть вопросов теории сетей Петри, связанных с выделением подкласса живых и безопасных сетей и их исследованием, близка к концепции самосинхронизации, причем несомненным достоинством сетевого подхода является возможность перехода от сети из указанного подкласса к самосинхронной схеме в обход процедуры кодирования.

Разработки Дж. Денниса [Dennis, 1970] и его коллег по проекту MAC в части создания машин, управляемых потоками данных, также концептуально близки к проблематике самосинхронных схем, но обих успехов в схемотехнической части, насколько выяснилось, достигнуто не было.

В 1972 г. изучение асинхронных триггерных схем натолкнуло проф. В. И. Варшавского на мысль о принудительном расщеплении переходного процесса в таких схемах на две фазы с целью фиксации окончания переходных процессов в каждой фазе. Работу [Варшавский, 1976] следует считать первой в мире монографией по проблеме самосинхронизации. Основной упор в ней делался на создание аperiodической схемотехники, что принципиально отличало подход авторов книги от подхода предшественников, например, описанного в [Cavagroc et al., 1974].

Физическое моделирование на уровне макетов вычислительных устройств на элементах малой степени интеграции подтвердило наличие положительных эффектов самосинхронной реализации — повышение быстродействия за счет работы по реальным задержкам элементов в 1,5—2 раза по сравнению с синхронным вариантом, а также дополнительный эффект — увеличение стабильности схем при вариациях напряжения питания и изменениях условий их функционирования.



Важнейшим результатом, полученным в последующем десятилетии [Варшавский, 1986], было установление свойства полной самопроверки самосинхронных схем относительно константных неисправностей элементов. Этот результат существенно с трех точек зрения:

самосинхронные схемы «безопасны» в том смысле, что их функционирование прекращается при возникновении неисправности и исключается возможность возникновения ложных событий;

неисправности самосинхронных схем могут быть просто локализованы при сопоставлении значения выходного сигнала с образцом; сигнал неисправности может быть использован для организации саморемонта;

самопроверяемость используется как средство функционального диагностирования.

В конце 70-х годов появились крупномасштабные программы по созданию больших, сверхбольших и ваферных интегральных схем — БИС, СБИС, ВИС. Профессора Ч. Сейц [Seitz, 1980] и Дж. Деннис [Dyant, 1980] выдвинули тезис о том, что синхронный подход не в состоянии гарантировать получение работоспособных СБИС в условиях субмикронной технологии. Дело в том, что по мере уменьшения размеров вентилях, повышения быстродействия, увеличения площади и «насыщенности» микросхем более остро проявляют себя технологические проблемы, связанные с обеспечением приемлемого процента выхода годных, рассеиванием мощности и существенным повышением величины задержек проводов относительно задержек вентилях. Эти проблемы неразрывно связаны с трудностями, обусловленными сложностью проектирования и тестирования СБИС, особенно заказных, а также с организацией межмодульных связей и синхронизации. Самосинхронный подход позволяет облегчить проектирование топологии схем и избежать необходимости генерации исчерпывающих тестов.

Проблема организации связей и взаимодействия модулей включает в себя как структурный, так и временной аспекты. Первый связан с тем, что в процессорных структурах с высокой степенью интеграции и параллелизма обмен сигналами неэффективен с точки зрения затрат площади, времени и мощности. Поэтому обмен стараются свести к локальным взаимодействиям (систолические и волнофронтные массивы) между соседними модулями. Второй аспект связан с синхронизацией и отражает тенденцию к смене ролей между задержками проводов и вентилях (особенно при проводных соединениях в диффузионном и поликремниевом слоях). Повышение относительной задержки провода замедляет работу синхронизатора, распределяющего синхросигналы по всей системе, так как частота такта должна быть снижена для компенсации перекоса локальных синхросигналов. Существует попытка паллиативного решения этой проблемы путем разбиения площади СБИС на так называемые зоны [Seitz, 1980], в пределах которых есть локальные часы. Однако такая попытка наталкивается на проблему устранения арбитражных аномалий и синхронизационных сбоев [Варшавский, 1986], возникающих при независимом тактировании зон.

Опыт проектирования самосинхронных структур продемонстрировал одновременно достоинства и недостатки канонического подхода. Пофрагментная трансляция спецификаций в модульные реализации, увеличивая скорость и уменьшая трудоемкость конструирования, оборачивается издержками в числе потребных элементов. Поэтому реально проектирование, особенно на уровне базисных узлов (ячеек библиотеки модулей), требует изобретательства. Сочетание неформальных шагов с формальными приемами связано с психологическими нагрузками дизайнеров, которые сильно возрастают при переучивании с целью освоения методологии самосинхронного подхода. По-видимому, нельзя будет полностью отказаться от традиционных видов схемотехники в пользу самосинхронной. Этими соображениями можно объяснить возникновение направлений, промежуточных между старыми и новыми. В соответствии с паллиативным (квазисамосинхронным) подходом устройство на некотором уровне не обязательно должно быть самосинхронным (например, оно может иметь локальное тактирование), но композиция таких устройств должна вести себя по установленным правилам, т. е. на следующем (верхнем) иерархическом уровне система обязана быть строго

самосинхронной. Уровень «чистоты» самосинхронности определяется представлениями проектировщика.

Такая идеология имеет под собой и другую аргументацию. К примеру, большинство имеющихся стандартных протоколов информационного обмена интерфейсов (как стандартных, так и новых перспективных типов, ориентированных на обмен типа «хэндшейкинг»<sup>1</sup>, естественного для концепции самосинхронизации) содержат временные функции, связанные с выходом из режимов по таймеру, бродкастингом, арбитражем, компенсацией перекосов и т. д. Даже «чистая» самосинхронность не решает всех проблем синхронизации для интерфейса, поэтому квазисамосинхронный подход оказывается более адекватным — важно лишь компенсировать собственно ошибки аппаратной реализации.

Учитывая все перечисленные проблемы, диктуемые практикой проектирования СБИС, были сделаны существенные попытки нового осмысления теории самосинхронизации. Прежде всего необходимо отметить новые подходы к определению того, что такое самосинхронная система.

Самосинхронная система определяется [Seitz, 1980] как самосинхронный элемент или «допустимая» композиция самосинхронных элементов. Это рекурсивное определение на абстрактном уровне не вызывает сомнений, но с конструктивной точки зрения мало что дает, поскольку понятие допустимой композиции может толковаться произвольно. Содержательно самосинхронные схемы можно толковать как композицию самосинхронных модулей, которые взаимодействуют через асинхронные протоколы. Однако основная сложность в определении понятия «допустимость»; оно всегда зависит от уровня детализации модели.

В работах группы Варшавского отмечается, что независимость от задержек и потому самопроверяемость на уровне логических элементов не означает наличия таких свойств на уровне транзисторов и проводных задержек. Таким образом, следует говорить не об абсолютной самосинхронности, а оговаривать заданный ее уровень.

Ввиду большой сложности проектов на уровне СБИС по-новому ставится теоретический вопрос о синтезе. Все большее внимание уделяется развитию методов блочного синтеза, в результате которых допустимость (или корректность) соединения самосинхронных элементов обеспечивается по построению — так называемая автокорректная реализация [Варшавский и др., 1988], которая осуществляется поблочной трансляцией элементов исходной спецификации в самосинхронные схемные «шаблоны». Однако при этом должна быть обеспечена проверка корректности исходной формальной спецификации по синтаксису и семантике.

Наряду с исследованиями в области традиционных для теории самосинхронизации формализмов наметилась тенденция к «анализу» других адекватных формализмов. Последние можно разделить на следующие категории: 1) теоретико-графовые модели глобального и событийного типа; 2) темпоральная (временная) логика; 3) модели семиотического типа; 4) модели, основанные на нотации языков программирования высокого уровня; 5) комбинированные модели.

Примерами моделей первого типа являются: сигнальные графы [Варшавский, 1986], диаграммы изменений [Варшавский и др., 1988], а также модели, связанные с параллельными граф-схемами алгоритмов (ГСА) и др. Принципы построения этих моделей однотипны: вершинам-событиям ставятся в соответствие изменения состояния управляющих и информационных переменных, а локальные отношения между парами отдельных вершин ставятся в соответствие тому или иному классу причинно-следственных связей между событиями в самосинхронной системе. Так, сигнальные сети Петри (сети Петри, переходы которых именуются значениями сигналов) существенно используют двудольность представляющего графа. Их анализ и приведение к полумодулярности связаны с необходимостью построения *графа достижимых маркировок* схемы и дальнейших трудоемких (экспоненциальных по сложности) манипуляций с ними.

<sup>1</sup> Хэндшейкинг (рукопожатие) — состояние готовности абонентов: одного — передавать, другого — принимать информацию.

Класс схем, представимых с помощью сигнальных графов, дистрибутивные схемы — уже, зато сложность их анализа при некоторых подходах может быть доведена до полиномиальной, к тому же схемы, как правило, получаются проще.

Диаграммы изменений функционально мощнее сигнальных графов, но сохраняют однодольность в отличие от сигнальных сетей Петри. Формализмы, базирующиеся на ГСА, представляют собой графы с множеством типов вершин, среди которых выделяются информационные, управляющие и смешанные, что сильно усложняет процедуру анализа.

В случае установления корректности спецификации на моделях первого типа, как правило, может быть осуществлен принцип автокорректирующей реализации. При этом речь идет только о синтезе управляющих схем.

Темпоральная логика также используется для задания и верификации самосинхронных структур [Malachi et al., 1981; Bochman, 1982]. Если при классическом маллеровском подходе для проверки полумодулярности используется гипотеза о конечности величин задержек элементов, то в [Mishra et al., 1985] обращено внимание на то, что менее жесткие гипотезы о характеристиках задержек (например, гипотеза «трех вторых»), а именно о том, что суммарная задержка трех элементов всегда превышает задержку двух, порождают качественно новые методы анализа.

Явным представителем модели семиотического типа является программная или символическая нотация. Переходя к моделям семиотического типа, следует прежде всего остановиться на бурно развивающемся направлении, называемом теорией трейсов (трассе) [Rem et al., 1983; Slepsecheut, 1983]. Под трейсами подразумеваются последовательности символов, отражающие возможные пути в диаграммах переходов или графах допустимых маркировок. По-видимому, трейсовый подход сыграет свою положительную роль скорее как промежуточный аппарат на определенных стадиях анализа моделей графового типа, хотя трудоемкость анализа на трейсах очень велика. Трейсы могут оказаться также весьма полезным механизмом порождения более сложного поведения из простых его форм при использовании композиционного подхода.

Модели четвертого типа фактически представляют собой языки программирования преимущественно с функциональной нотацией — типа Оккам, Алгол-68, Ада [Kelem, 1985] и др. Так же, как при кремниевой компиляции, операторные конструкции таких языков переводятся в темплейты (шаблоны), каждому из которых соответствует самосинхронный модуль. Такой подход ограничен из-за отсутствия поддержки соответствующим набором автоматизированных интеллектуальных средств проектирования и библиотек схемотехнических решений и приводит к малоэффективным реализациям. Он применим пока лишь к объектам с примитивной структурой управления, и, возможно, развитыми информационными потоками.

Пятый подход характеризуется тем, что от языков высокого уровня осуществляется переход к некоторой модели первого типа, принятой в системе в качестве объектного механизма, например к сети Петри, интерпретированной потоком данных. Такое объектное описание анализируется на корректность, после чего осуществляется обычная процедура перехода к самосинхронной схеме. К сожалению, этот подход узко проблемно-ориентирован, в основном это способы потоковой обработки данных в сигнальных процессорах.

Развитие теоретических схемотехнических исследований, основные направления которых отражены выше, должно быть подкреплено практическими работами, отвечающими запросам современной вычислительной техники. Известные факты внедрения — от самосинхронных схем узлов компьютеров до суперкомпьютера Cosmic Cube — приведены в обзоре [Барни, 1985]. Дальнейшее распространение практики самосинхронных разработок зависит также от уровня инструментального обеспечения широкого круга разработчиков. В связи с этим предпринимаются активные попытки создания и тиражирования через информационные сети связи библиотек типовых схемотехнических решений как в рамках чисто самосинхронной, так и квазисамосинхронной идеологии.

**Три вида динамических моделей, адекватных задачам проектирования самосинхронных схем.**

**Сеть Петри** [Котов, 1984; Питерсон, 1984; Розенблум, 1983] — это двудольный ориентированный граф  $\langle P, T, E, M_0 \rangle$  с двумя типами вершин — позициями из множества  $P$ , переходами из множества  $T$  (изображаются соответствием кружками и черточками), дугами  $E \subseteq (P \times T) \cup (T \times P)$ , соединяющими позиции с переходами и переходы с позициями, и начальной маркировкой  $M_0, M: P \rightarrow N$ , где  $N$  — множество неотрицательных целых чисел. Маркировка представляется вектором, число компонентов которого равно числу позиций, а значение каждого компонента, отличное от нуля, соответствует числу маркеров — жирных точек, помещаемых внутри позиции. Динамика в сети Петри описывается перемещением маркеров по сети в соответствии с соглашением о правиле срабатывания перехода, носящем локальный характер. Переход считается возбужденным, если каждая из его входных позиций содержит по крайней мере по одному маркеру. Через некоторое конечное (заранее не заданное) время возбужденный переход срабатывает, в результате чего из каждой его входной позиции изымается по одному маркеру, а в каждую его выходную позицию добавляется по одному маркеру (изъятие и добавление маркеров осуществляется одновременно). Сеть Петри функционирует, переходя от одной маркировки к другой посредством срабатываний возбужденных переходов. Поэтому ее поведение характеризует диаграмма маркировок — оргграф, вершинами которого являются маркировки, а дуги помечены наименованиями срабатывающих переходов.

Классическими массовыми задачами теории сетей Петри являются анализ на:

- достижимость (возможно ли достижение некоторой исходно заданной маркировки  $M^*$  из начальной  $M_0$  в данной сети, т. е. содержится ли  $M^*$  в диаграмме маркировок);

- живость, или активность (возможно ли в принципе срабатывание любого перехода данной сети);

- одновременность (возможно ли параллельное срабатывание нескольких переходов сети);

- ограниченность (может ли в процессе функционирования сети в какой-либо ее позиции наблюдаться число маркеров, не превышающее заданное; в неограниченной сети в позициях может накапливаться бесконечное число маркеров);

- безопасность (частный случай ограниченности, когда все достижимые маркировки представляют собой булевы векторы);

- устойчивость (когда срабатывание одного перехода не может вызвать сию же возбуждения другого) и др.

Доказана алгоритмическая разрешимость этих задач, в общем случае являющихся экспоненциальными по сложности. В приложениях чаще всего используются определенные подклассы сетей Петри — автономные (живые, в которых из любой маркировки достижима любая другая), безопасные и бесконфликтные, допускающие естественную интерпретацию.

**Маркированные графы** — это такие сети Петри, в которых ни одна позиция не содержит более одного входного и одного выходного перехода, что позволяет изображать их в виде однодольных оргграфов  $\langle V, E \subseteq V \times V, M_0 \rangle$ , полученных из сетей исключением позиций (перечеркиванием их дугами, взвешенными имеющимися в позициях маркерами) и заменой переходов вершинами. Правило срабатывания возбужденных вершин маркированного графа состоит в изъятии одного маркера из каждой ее входной дуги и добавлении одного маркера на каждую ее выходную дугу.

Иногда полезно использовать интерпретацию модели маркированного графа — **сигнальный граф** [Варшавский, 1986; Rosenblum et al., 1985]. Для его задания требуется ввести множество булевых переменных (сигналов)  $Z = \{z_1, \dots, z_n\}$ , множество их изменений  $\delta Z = \{+z_i, -z_i, i=1, \dots, n$ , которые означают соответственно переходы 0-1 и 1-0 сигналов  $z_i$ , и функцию разметки  $\delta: V \rightarrow \delta Z$ , ставящей в соответствие вершинам маркированного графа изменения сигналов.

Алгоритмы анализа сигнальных графов, заданных на автономных, живых и безопасных маркированных графах, имеют полиномиальную сложность порядка  $O(|V|^3)$ , зависящую от числа вершин графа. Это заставляет считать соответствующий аппарат более предпочтительным по сравнению с анализом по диаграммам Маллера, имеющим экспоненциальную сложность. Цель такого анализа — установление наличия некоторых специфических свойств сигнальных графов, позволяющих использовать их как спецификацию для последующего синтеза самосинхронных схем.

### Пример

Описываемый ниже пример не относится непосредственно к интеллектуальным системам. Однако предлагаемый подход оказывается полезным при проектировании средств аппаратной поддержки интеллектуальных систем, при построении вычислительных средств ЭВМ 5-го поколения и при организации управления в робототехнических системах. Особенно привлекательно использование самосинхронного подхода в роботике при описании синергий формирования локомоций как блоков динамического стереотипа, которые могут быть схемно реализованы, что значительно разгружает центральное управление роботами, высвобождая его от функций контроля за действиями исполнительных механизмов. Существенными чертами дискретных систем управления, для которых удобно использовать самосинхронный подход, являются: 1) наличие у процессов явно выраженных фаз, в конце каждой из которых выдается информация о ее завершении, 2) согласованность: осуществление перехода к следующей фазе только после получения сигнала о завершении предыдущей, 3) параллельность: возможность одновременных событий в системе, 4) асинхронность: отсутствие ограничений на относительную длительность осуществления переходов из состояния в состояние, зависящих от многочисленных неконтролируемых факторов.

Пусть требуется синтезировать схему управления движением двух передвигающихся по рельсам вагонеток 1 и 2, осуществляющих автоматическую загрузку, транспортировку и разгрузку сыпучего груза (например, песка). Эскиз сцены представлен на рис. 4.9.

Вагонетка стоящая в пункте *B* непосредственно под воронкой, загружается песком через воронку. заслонка (3) которой открыта на время, пока песок в вагонетке не достигнет верхнего уровня, отмечаемого датчиком. После этого заслонка закрывается.

Разгрузка вагонетки, стоящей в пункте *G*, выполняется при открытии находящегося на дне вагонетки люка (*Л*). Когда песка в вагонетке не остается, что отмечается датчиком нижнего уровня, люк закрывается. Груз сыпается в накопительную емкость.

Закончив загрузку (разгрузку), вагонетка должна переехать в пункт разгрузки (загрузки). Движение осуществляется по двухколейной железнодорожной ветке, но ближайшие подходы к терминальным пунктам являются однопутными. Вагонетки могут одновременно передвигаться либо слева направо по ветке *L2*, *R1*, либо справа налево по ветке *R2*, *L1*, для чего каждая из них снабжена реверсивным двигателем. Естественно, что управление должно содержать механизмы, препятствующие столкновениям вагонеток.

Из соображений, которые станут понятными позже, на рис. 4.9 символами *L* и *R* отмечены две промежуточные станции, в которых требуется останавливать вагонетки, если возникает реальная опасность их столкновения, а также четыре семафора: *L1* (на участке *L—B*), *L2(B—R)*, *R1(R—G)*, *R2(G—L)*. Семафоры можно толковать как предикаты, принимающие значения 1, если путь открыт, и 0 в противном случае. Кроме того, имеет смысл ввести другие предикаты и пропозициональные переменные, необходимые для адекватного описания функционирования системы. Это:

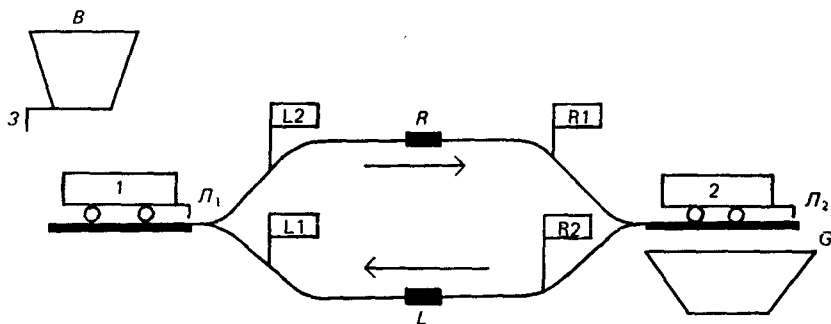


Рис. 4.9. Схема перемещения вагонеток

переменные  $x_B, x_R, x_G, x_L$ , моделирующие датчики, установленные в пунктах  $B, R, G, L$  и пути и принимающие значение 1/0, если какая-либо из вагонеток находится в данном пункте/вне его;

переменные  $x_i^1, x_i^2, i = 1, 2$ , моделирующие датчики верхнего и нижнего положений сыпучего груза, которыми должна быть оборудована каждая вагонетка;  $x_i^1 = 1/0$ , если вагонетка полна/неполна;  $x_i^2 = 1/0$ , если вагонетка непуста/пуста;

переменная  $y_z$ , моделирующая состояние заслонки воронки,  $y_z = 1/0$ , если заслонка открыта/закрыта;

предикаты  $y_R^i, y_L^i, i = 1, 2$ , моделирующие состояние исполнительного двигателя  $i$ -й вагонетки,  $y_R^i = 1/0$ , если двигатель включен для движения вправо/остановлен,  $y_L^i = 1/0$ , если двигатель включен для движения влево/остановлен;

переменные  $y_i^1, i = 1, 2$ , моделирующие положение люка  $i$ -й вагонетки,  $y_i^1 = 1/0$ , если люк открыт/закрыт.

Теперь имеет смысл формально задать процесс  $P_i$  движения одной ( $i$ -й) вагонетки,  $i = 1, 2$ . Для этого выберем изобразительные средства сетей Петри.

Процесс, изображенный на рис. 4.10,а, является последовательным и состоит в циклическом прохождении пунктов  $B$  (загрузка),  $R, G$  (разгрузка) и  $L$  (этими символами помечены переходы сети Петри). Наличие маркера во входной (выходной) позиции перехода трактуется как движение вагонетки из предыдущего (данного) пункта в данный (следующий), а факт срабатывания перехода  $B, R, G, L$  — как окончание загрузки, прохождение пункта  $R$ , окончание разгрузки, прохождение пункта  $L$  соответственно.

Общая модель синхронизации процессов  $P1$  и  $P2$  движения обеих вагонеток, одна из которых исходно находится в пункте загрузки, а вторая — в пункте разгрузки, показана на рис. 4.10,б. Синхронизация осуществляется с помощью позиций, помеченных символами  $L1, L2, R1, R2$ , механизм реализации синхронизации эквивалентен использованию семафоров Э. Дейкстры [Питерсон, 1984; Розенблум, 1983]. Анализ представленной сети Петри показывает, что она является живой, безопасной и устойчивой (невозможно одновременно возбуждение одноименных переходов, т. е. столкновение вагонеток в каком-либо пункте пути). Можно показать, что уменьшение числа семафоров до двух (например, попытка использовать только семафоры  $L2$  и  $R1$ ) приводит к нарушению безопасности и устойчивости. Доказательство устойчивости сети Петри, представленной на рис. 4.10, следует из рис. 4.11,а, на котором показан граф достижимых маркировок. Начальной маркировке  $M_0$  (рис. 4.11,а) соответствует маркировка, представленная на рис. 4.10. Видно, что в этом варианте осуществляется достаточно жесткая синхронизация: выезд вагонетки из данного пунк-

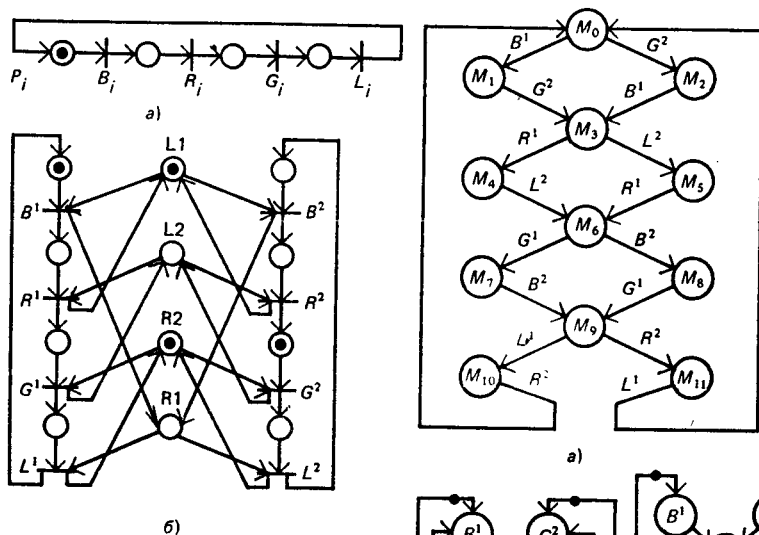
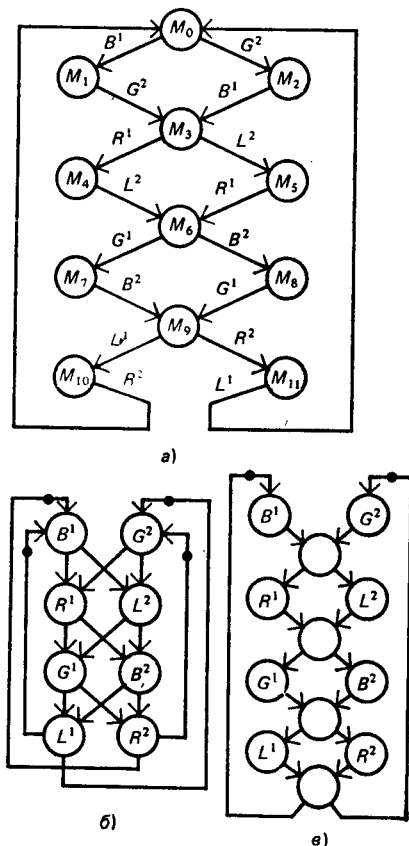


Рис. 4.10. Сеть Петри, описывающая перемещение 1-й вагонетки (а) и двух вагонеток (б)

Рис. 4.11. Анализ живости, безопасности и устойчивости сети Петри с помощью графа достижимых маркировок (а), маркированного (б) и сигнального (в) графа



та  $B, R, G, L$  возможен только в том случае, когда вторая вагонетка готова к выезду из диаметрально расположенных по отношению к нему пунктов  $G, L, B, R$  соответственно. Такое же поведение имеет маркированный граф, представленный на рис. 4.11,б. Чтобы убедиться в этом, достаточно сравнить рис. 4.10,б и 4.11,б.

Наконец, следует отметить, что сигнальный граф, показанный на рис. 4.11,в, который содержит по сравнению с графом на рис. 4.11,б лишние непомеченные вершины, эквивалентен ему, ибо последовательности срабатываний помеченных вершин в обоих графах одинаковы.

Далее следует осуществить детализацию подпроцессов, происходящих «на фоне» движения; включение и выключение двигателей, открытие и закрытие заслонок и люков и пр. Такие подпроцессы удобнее всего (имея в виду не только способ спецификации, но и дальнейший переход к синтезу схемы управления) задавать на языке сигнальных графов.

На рис. 4.12,а приведен фрагмент сигнального графа (без маркировки дуг), описывающего подпроцесс  $B^i(G^i)$ , включающий события, связанные с подъездом  $i$ -й вагонетки к пункту  $B(G)$  и загрузкой (разгрузкой) песка. Семантика (интерпретация вершин) ясна из сопровождающих надписей. Знаком плюс отмечены события типа включения, наполнения или открытия исполнительных

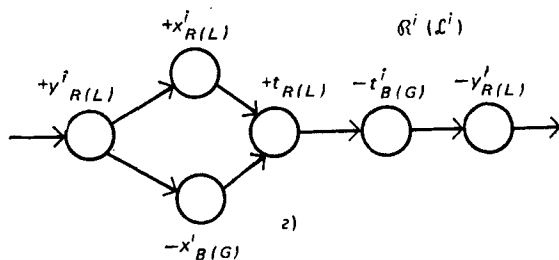
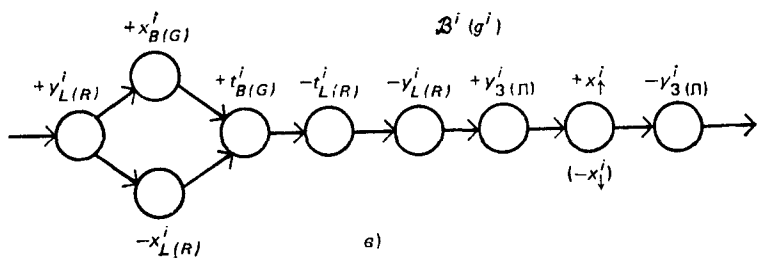
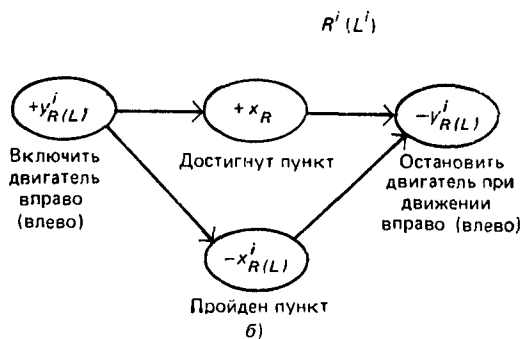
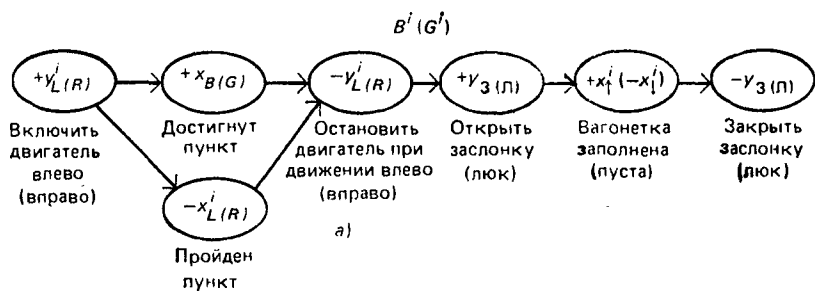


Рис. 4.12. Фрагменты движения вагонеток



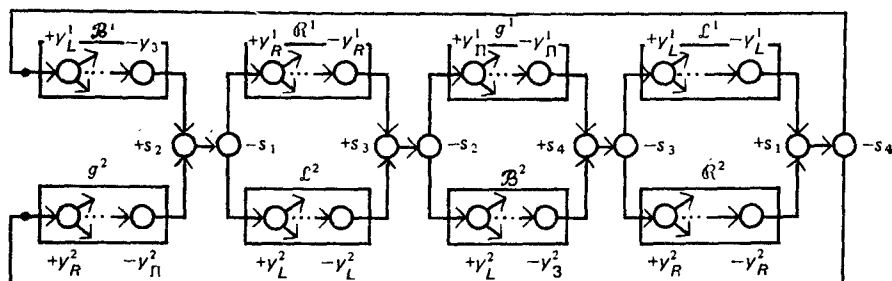


Рис. 4.13. Блочное изображение сигнального графа

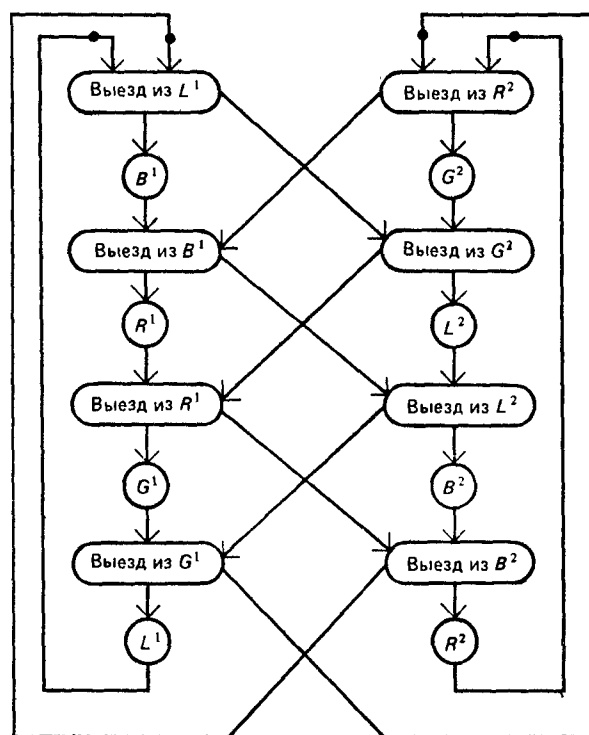


Рис. 4.14. Маркированный граф для измененного выезда

механизмов, знаком минус — события типа их выключения, остановки, закрытия и пр. На рис. 4.12,б описан подпроцесс  $R^i(L^i)$  подъезда  $i$ -й вагонетки к пункту  $R(L)$ . Эти фрагменты непосредственно реализовать нельзя из-за того, что по семантике требуется введение дополнительных переменных  $t_{R(L)}^i$  и  $t_{B(G)}^i$ , запоминающих текущие места пребывания  $i$ -й вагонетки на соответствующих участках пути; без них не удастся координировать совместное движение вагонеток. Поэтому от фрагментов процессов  $B^i(G^i)$  и  $R^i(L^i)$ , показанных на

рис. 4.12,а и б, осуществлен переход к их модификациям  $\mathcal{B}^i(\mathcal{B}^i)$  и  $\mathcal{R}^i(\mathcal{R}^i)$ , представленным соответственно на рис. 4.12,в и г.

Далее требуется перейти к окончательному представлению сигнального графа (рис. 4.13). В нем подпроцессы синхронизируются по типу рис. 4.11,в, но вместо каждой непомеченной вершины граф содержит пару вершин, обозначенных символами  $\pm s_j$ ,  $j=1, \dots, 4$ . Введение этих дополнительных переменных обосновывается синтаксическими требованиями к сигнальному графу как спецификации самосинхронной схемы [Варшавский и др., 1988]. Адекватный анализ графа, показанного на рис. 4.13, свидетельствует о том, что он удовлетворяет этим требованиям.

Полученная спецификация не является, естественно, единственной. Можно предложить и другие, в том числе с большей степенью параллелизма (их существование вызывается грубостью вербального описания задачи, и возможность той или иной детализации спецификации зависит от знаний и интуиции разработчика). Например, можно изменить условие выезда вагонетки. Пусть выезд ее из данного пункта  $B, R, G, L$  возможен после выезда второй вагонетки из следующего по направлению движения пункта  $G, L, B, R$ . Соответствующий маркированный граф показан на рис. 4.14. Для достижения более высокого уровня параллелизма по сравнению с графом, представленным на рис. 4.11,б, здесь потребовалось расщепить каждый процесс на два подпроцесса: а) достижения пункта и работы в нем; б) выезда из пункта. Переход к самосинхронной реализации также не является однозначным. Здесь выбран стандартный, канонический подход, не требующий от проектировщика глубоких познаний в области теории самосинхронных схем. Необходимо лишь владеть приемами так называемой парафазной реализации [Варшавский, 1986], при которой каждой переменной ставится в соответствие RS-триггер, поведение которого задается автоматным уравнением  $t = S \vee R \bar{z}$ . Функции возбуждения  $S$  и  $R$ , фигурирующие в этом уравнении, должны быть: 1) ортогональными, т. е. должно выполняться условие  $SR=0$  (либо формально, либо вследствие неодновременности), 2) представлены в сокращенной, а не минимальной нормальной форме.

Приведем эти функции для каждой из фигурирующих в сигнальном графе переменной:

$$y_{R(L)}^i: S = t_{B(G)}^i \bar{s}_1(3) \vee t_{R(L)}^i \bar{s}_2(4), \quad R = \bar{t}_{B(G)}^i s_2(4) \vee \bar{t}_{R(L)}^i s_3(1);$$

$$y_3: S = \bar{y}_L^i t_B^i \bar{x}_\downarrow^i \vee \bar{y}_L^i t_B^i x_\downarrow^i, \quad R = x_\uparrow^i x_\downarrow^i s_1 \vee x_\uparrow^i x_\downarrow^i s_3;$$

$$y_n^i: S = \bar{y}_R^i t_G^i x_\uparrow^i, \quad R = \bar{x}_\downarrow^i \bar{x}_\uparrow^i;$$

$$t_{R(L, G, R)}: S = x_{R(L, G, B)} \bar{x}_{B(G, R, I)} t_{B(G, R, L)}^i, \quad R = t_{G(B, L, R)};$$

$$s_1: S = \bar{y}_L^i t_L^i \bar{y}_R^i t_R^i s_4, \quad R = s_2;$$

$$s_2: S = \bar{y}_3 x_\uparrow^i \bar{y}_n^i x_\downarrow^i s_1, \quad R = s_3;$$

$$s_3: S = \bar{y}_R^i t_R^i \bar{y}_L^i t_L^i s_2, \quad R = s_4;$$

$$s_4: S = \bar{y}_n^i \bar{x}_\downarrow^i \bar{y}_3 x_\uparrow^i s_3, \quad R = s_1.$$

Схемы триггеров, реализующих типовые функции приведенной системы, представлены на рис. 4.15. Все триггеры с парафазным выходом ( $z, \bar{z}$ ) имеют транзитное состояние  $(0, 0)$ . На рис. 4.15,а показан триггер, который при прибытии вагонетки в пункт  $R$  (срабатывании датчика, выдающего на вход триггера сигнал  $R$ ) устанавливается в состояние  $x_R=1$ ,  $\bar{x}_R=0$ , а после отъезда вагонетки из пункта  $R$  — в состояние  $x_R=0$ ,  $\bar{x}_R=1$ . Аналогично реализуются триггеры  $x_L, x_G, x_B, x_\uparrow^i, x_\downarrow^i$ . Триггеры  $y_R^i, y_L^i$  реализуются по типу рис. 4.15,б,  $s_1-s_4$  — по типу рис. 4.15,в.  $u_n^i$  — по типу рис. 4.15,г;  $t_R, t_L$  — по типу рис. 4.15,д;  $y_3^i$  — по типу рис. 4.15,е.

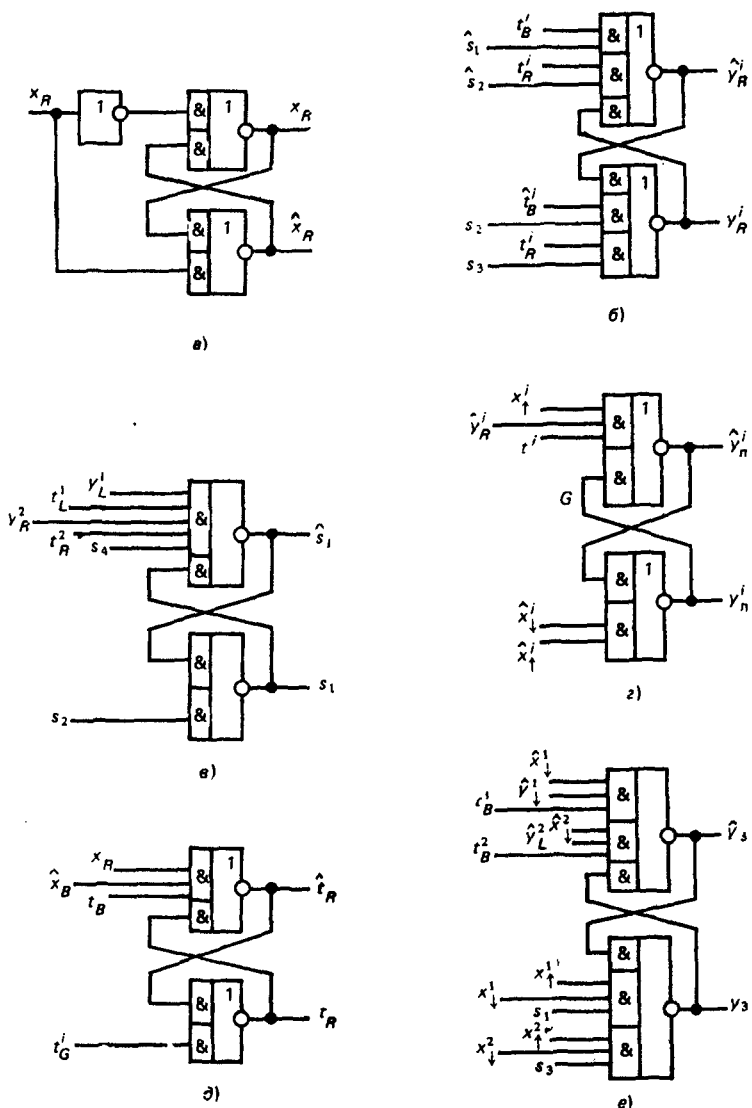


Рис. 4.15. Реализация типовых функций системы

Общая схема управления содержит всего 22 триггера, которые соединяются в соответствии с системой типовых функций, причем вместо переменных, содержащихся в правых частях уравнений в прямом (инверсном) виде, используется выход плеча, не содержащего (содержащего) крышку.

Важно отметить присущую асинхронной схемотехнике характерную особенность. Операционные модули систем обычно выполняются таким образом, что каждый из них помимо информационных входов и выходов снабжен па-

рой дополнительных управляющих сигналов — запроса и ответа. Поэтому функционирование такого модуля моделирует элемент задержки, входом которого является сигнал запроса, а выходом — сигнал ответа. Предположение о конечности величин задержек позволяет осуществить композицию операционных модулей и схемы управления модулей в разрывы проводов последней. При этом сохраняется та же координация (временная последовательность) включения и выключения модулей, которая имеет место для элементов схемы управления.

### **Перспективы**

Самосинхронная схемотехника может оказаться перспективным направлением в развитии вычислительной техники и, следовательно, аппаратных средств поддержки интеллектуальных систем. В пользу этой гипотезы — множество разнообразных факторов, против, по-видимому, только один: необходимость преодоления психологического отторжения концепции, не очень знакомой большинству проектировщиков и достаточно сложной по сравнению с общепринятой. Можно ожидать, что после создания эффективных САПР появятся изделия, соответствующие новинкам теоретических исследований в области самосинхронизации, и возникнут рыночные тенденции, стимулирующие массовый выпуск самосинхронной техники.

Что касается ближайшей перспективы, то направления фундаментальных исследований более или менее ясны:

1) продолжение углубления теории самосинхронных схем для построения результатов по линии обеспечения задач САПР;

2) переход на решение задач синтеза самосинхронных схем с уровня функциональных элементов на уровень транзисторов для различных интегральных технологий;

3) решение задач повышения уровня выхода годных вафер-ИС за счет введения избыточности, механизмов локализации неисправностей и организации саморемонта для парирования технологических дефектов;

4) формирование более четких критериев определения самосинхронных структур, в отсутствие которых затруднена оценка результатов и реализации: с точки зрения «чистой» теории большинство выпускаемых схем, объявленных создателями самосинхронными, не принадлежат к классу независимых от скорости, а следовательно, не обладают частью полезных свойств, из которых наиболее значительным является самопроверяемость.

Используемые при реализации самосинхронных схем модели (сети Петри, сигнальные графы и др.) могут с успехом применяться при решении многих задач создания интеллектуальных систем. Так, сети Петри служат адекватным средством описания и анализа продукционных и сетевых моделей, дедуктивного вывода и т. п. [Вагин и др., 1987].

## **Глава 5.**

### **Специализированные процессоры для интеллектуальных систем**

#### **5.1. Машины баз данных**

*М. М. Гилула, Л. А. Калинин, С. К. Ландо, В. М. Рывкин*

#### **Назначение машин баз данных**

Основы современной информационной технологии составляют базы данных (БД) и системы управления базами данных (СУБД), роль которых как единого средства хранения, обработки и доступа к большим объемам информации

постоянно возрастает. При этом существенным является постоянное повышение объемов информации, хранимой в БД, что влечет за собой требование увеличения производительности таких систем. Резко возрастает также в разнообразных применениях спрос на интеллектуальный доступ к информации. Это особенно проявляется при организации логической обработки информации в системах баз знаний, на основе которых создаются современные экспертные системы.

Быстрое развитие потребностей применений БД выдвигает новые требования к СУБД:

- поддержка широкого спектра типов представляемых данных и операций над ними (включая фактографические, документальные, картинно-графические данные);

- естественные и эффективные представления в БД разнообразных отношений между объектами предметных областей (например, пространственно-временных с обеспечением визуализации данных);

- поддержка непротиворечивости данных и реализация дедуктивных БД; обеспечение целостности БД в широком диапазоне разнообразных предметных областей и операционных обстановок;

- управление распределенными БД, интеграция неоднородных баз данных; существенное повышение надежности функционирования БД. Вместе с тем традиционная программная реализация многочисленных функций современных СУБД на ЭВМ общего назначения приводит к громоздким и непроизводительным системам с недостаточно высокой надежностью. Тем более затруднительным оказывается наращивание программных средств, обеспечивающих перечисленные выше требования. Это обусловлено рядом причин:

- фон-неймановская архитектура ЭВМ неадекватна требованиям СУБД, в частности реализации поиска, обновления, защиты данных, обработки транзакций только программным способом неэффективны как по производительности, так и по стоимости;

- многоуровневое и сложное программное обеспечение СУБД снижает эффективность и надежность функционирования БД;

- универсальная ЭВМ оказывается перегруженной функциями управления базами данных, что снижает эффективность функционирования собственно прикладных систем;

- централизация и интеграция данных в сетях персональных и профессиональных ЭВМ нереализуема с приемлемой стоимостью без включения в состав сетей специализированных ЭВМ для поддержки функций СУБД.

Эти соображения приводят к мысли о необходимости создания специализированных автономных информационных систем [Magyanski, 1980], ориентированных исключительно на реализацию функций СУБД. Однако системы, реализованные на обычной универсальной мини- или микроЭВМ, не способны полностью решить указанные проблемы. Необходим поиск новых архитектурных и аппаратных решений. Исследования в этом направлении привели к появлению проектов и действующих прототипов машин баз данных, которые наряду с самостоятельным назначением составляют также основу вычислительных систем 5-го поколения [Miyakami, 1985]. Машинной баз данных (МБД) принято называть аппаратно-программный мультимикропроцессорный комплекс, предназначенный для выполнения всех или некоторых функций СУБД.

Такие свойства реляционной модели данных, как возможность расчленения отношений на непересекающиеся группы, возможность массовой и параллельной обработки, простота и независимость данных в этой модели, а также наличие развитой теории реляционных баз данных [Майер, 1986] и аппарата сведения к реляционной других моделей данных [Калининченко, 1983] обусловили разработку МБД, ориентированных в основном на поддержку реляционных баз данных [Hsiao, 1983; Leilich, 1983; DeWitt, 1985; Sood, 1985; Ozkanal, 1986]. В настоящее время очевидна правильность такого выбора в связи с установленной возможностью оперировать объектами баз знаний на реляционном концептуальном уровне посредством операций реляционной алгебры.

Список проектов МБД

№ п/п	Проект	Автор или фирма	Страна	Год
1	CAFS	BABB E.	США	1979
2	DBC/1012	TERADATA INC.	США	1983
3	DBMAC	MISSIKOFF M. ET AL.	Италия	1983
4	DELTA	MOTO-OKA T., ET AL (ICOT)	Япония	1981
5	DSDBC	TANAKA Y. ET AL	Япония	1980
6	GAMMA	DEWITT D. ET AL	США	1986
7	GRACE	KITSUREGAWA M., MOTO-OKA T.	Япония	1983
8	IDM500	EPSTEIN R. (BRITTON LEE INC.)	США	1982
9	LUCAS	LINDH G.	Швеция	1986
10	MBDS	HSIAO D. ET AL	США	1983
11	MIRDM	QADAH G., IRANI K.	США	1985
12	MRDC	TANAKA Y.	Япония	1984
13	NODD	BIC L.	США	1983
14	PPRQP	WON KIM ET AL.	США	1980
15	RAP3	OZKARAHAN E. ET AL	США	1983
16	RDBM	AUER H., HELL M., LEILICH H.	ФРГ	1981
17	RS310	HAWTORN P. (BRITTON LEE)	США	1985
18	SABRE	VOLDURIEZ P., GARDARIN D.	Франция	1982
19	SIMD	RICHAER K. ET AL.	ЧССР	1982
20	SPIRIT3	KAMIBAYASHI N. ET AL	Япония	1981
21	SURE	LEILICH H. ET AL.	ФРГ	1978
22	VERSO	GAMERMAN S., SCHOOL M.	Франция	1982

Первые публикации по МБД появились в 1974 г., сейчас можно назвать более 50 проектов, некоторые уже реализованы в виде промышленных прототипов и являются коммерческими изделиями [Калниниченко, 1987]. В табл. 5.1 приведен список наиболее интересных проектов и промышленных прототипов МБД. Исследования по аппаратурной поддержке операций над базами данных проводятся и в нашей стране [Задыхайло, 1979; Суворов, 1985]. Основными критериями для оценки того или иного проекта являются полнота выполняемых функций СУБД и ожидаемое повышение производительности при их выполнении. Это одинаково важно как для МБД, функционирующих совместно с главной ЭВМ в составе единой вычислительной системы, так и для МБД, являющейся узлом локальной сети (data computer). Во всех современных проектах и коммерческих МБД реализован полный объем функций СУБД. Повысить производительность, учитывая ограниченные скоростные характеристики современной элементной базы, можно только структурными методами (за счет структурного распараллеливания). В силу этого МБД являются специализированными параллельными вычислительными системами, и при их проектировании требуются единая методология сравнения и четкие критерии оценки производительности. В настоящее время ведутся интенсивные исследования в этой области, обширная библиография приведена в [Калниниченко, 1987].

Основными техническими приемами, применяемыми в структурных методах повышения производительности МБД, являются следующие:

использование многоканальных устройств массовой памяти (УМП) со встроенными в аппаратуру каналов процессорами поиска и фильтрации для уменьшения объемов перекачиваемых данных из УМП в обрабатывающие подсистемы;

использование буферизации между основной памятью обрабатывающих процессоров и УМП, которая не только сглаживает разницу в скоростях обработки данных и чтения их в УМП, но и уменьшает частоту обращения к УМП;

сегментация данных в УМП, которая увеличивает локальность доступа и улучшает эффект двух предыдущих методов; с этой целью предполагается развитие мультнарибутной кластеризации и индексации данных в УМП и аппаратная их поддержка;

использование ассоциативной памяти в качестве буферной и соответствующих алгоритмов обработки данных;

развитие подсистем опережающей выборки данных в буферную память (стадирование данных) и оптимизация алгоритмов управления виртуальным пространством данных;

реализации режимов параллельной интерпретации каждой операции над БД (горизонтальный параллелизм типа SIMD) и режимов конвейерной и поточковой обработки не только операций, но и транзакций в целом;

функциональная специализация процессоров обработки и их аппаратная реализация в виде СБИС.

### Основные направления развития структур МБД

Можно выделить два обобщенных направления, в которых ведутся исследования по структурным методам повышения производительности МБД: многопроцессорные неоднородные (МН) и сетевые МБД. На рис. 5.1 приведены обобщенные топологические схемы таких МБД. Частным случаем МН МБД можно считать коммерческие МБД IBM-500, RS-310, IDBP 86/440, топологическая схема которых приведена на рис. 5.2.

К МН МБД можно отнести большинство современных проектов МБД, таких как DELTA, GRACE, DSDBS, MPDC, SABRE и др. Основными особенностями МН МБД являются следующие.

1. Наличие нескольких уровней обработки данных, в частности, трех основных:

селекция и первичная фильтрация данных непосредственно в контрольных устройствах массовой памяти;

вторичная обработка, заключающаяся в реализации операций реляционной алгебры над вспомогательными отношениями, полученными на первом этапе;

структурная обработка или обработка метаданных, заключающаяся в поддержке вспомогательных структур данных (индексация, мультнарибутная кластеризация).

2. Наличие системной буферной памяти (СБП) между первыми двумя уровнями обработки, в которой помещаются отношения или вспомогательные структуры данных, полученные на первом уровне обработки. Такая архитектура предполагает наличие опережающей выборки и подкачки данных из уровня первичной обработки (стадирование). СБП при этом обязательно должна быть двух- или более портовой.

3. Наличие функционального параллелизма, при котором различные функции первичной и вторичной обработки реализуются на физически распределенной аппаратуре. При этом часть функциональных устройств реализуется на универсальных микропроцессорах, часть в виде спецаппаратуры (например, заказных СБИС). Функциональный параллелизм позволяет реализовать конвейерное выполнение транзакций и отдельных запросов. В более общих случаях для увеличения производительности допускается дублирование функциональных процессоров на наиболее трудоемких операциях.

В качестве наиболее типичных примеров таких МН МБД можно рассмотреть DELTA и GRACE. Японский проект МБД (рис. 5.3) лежит в основе вычислительной системы 5-го поколения [Shibuyama, 1984]. Действующий в настоящее время прототип состоит из двух подсистем:

подсистемы вторичной обработки в составе четырех реляционных процессоров (РП), одного процессора управления (УП), одного коммуникационного

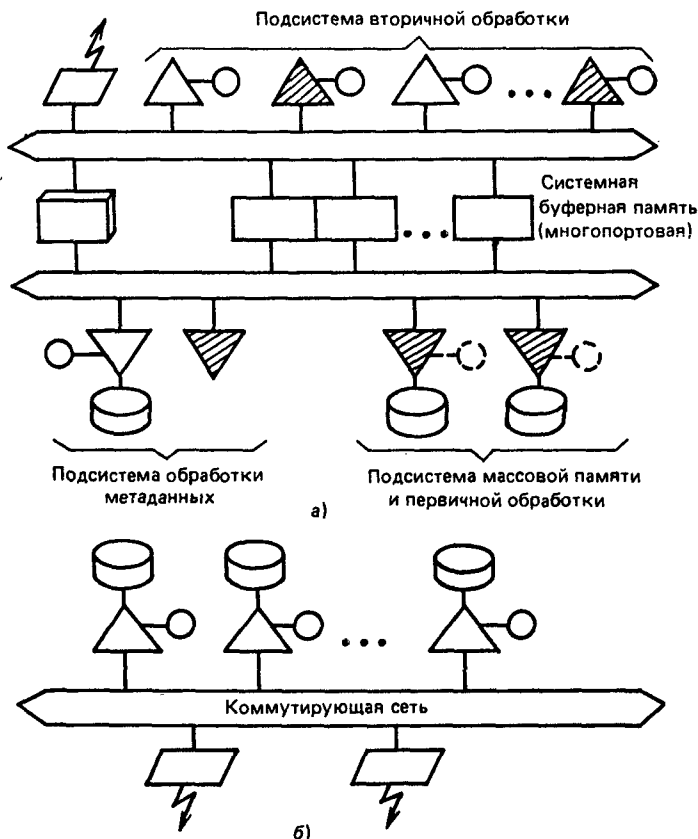


Рис. 5.1. Топология двух классов МБД:

а — многопроцессорные неоднородные МБД с несколькими уровнями обработки;  
б — сетевые МБД

процессора (КП) и одного процессора технического обслуживания (ПТО), выполняющего функции диагностики системы, поддержки БД, связи с оператором и т. п.;

подсистемы иерархической памяти (ИП), содержащей системную буферную память (электронный кэш-диск емкостью 128 Мбайт), массовую память с восемью НМД (с контроллером магнитного диска КМД) общей емкостью 20 Гбайт и четырьмя НМЛ (с контроллером магнитной ленты — КМЛ), а также универсальную микроЭВМ в качестве управляющего процессора иерархической памяти (УПИП) и процессора ввода-вывода (ПВВ). Связь между подсистемами осуществляется высокоскоростным каналом со стандартным интерфейсом со скоростью передачи до 3 Мбайт/с. Все процессоры подсистемы вторичной обработки подключаются к этому каналу посредством ПВВ через специальные адаптеры иерархической памяти (АИП).

Основным функциональным узлом МБД DELTA является реляционный процессор (РП) баз данных, назначение которого — выполнение операций реляционной алгебры над отношениями произвольного объема с высокой производительностью. Каждый из четырех РП может выполнять отдельную операцию



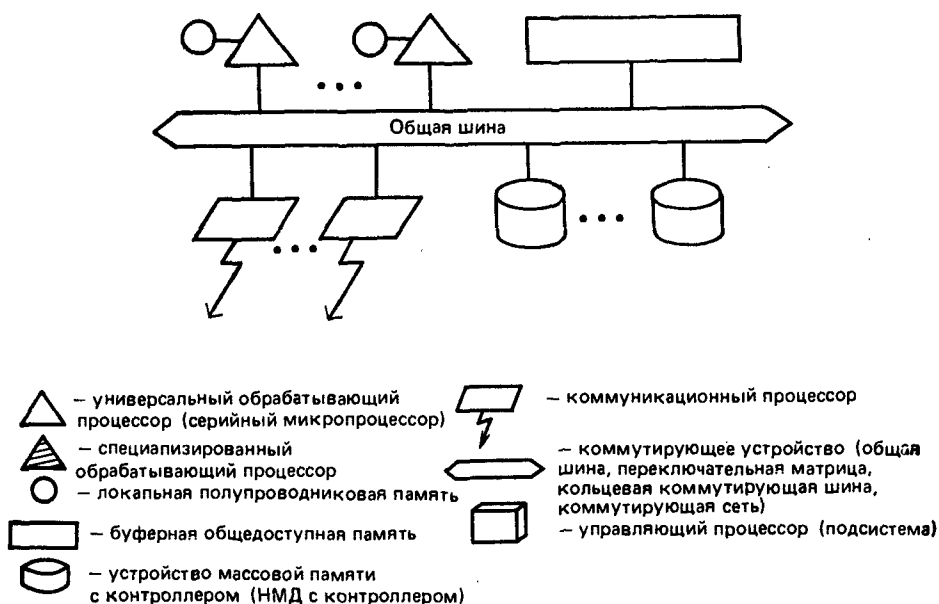


Рис. 5.2. Топология коммерческих МБД.

реляционной алгебры независимо от других или все они могут выполнять одну операцию параллельно (например, сортировку отношений в ИП). РП имеет регулярную структуру (см. рис. 5.3) для облегчения его реализации в виде СБИС. Кроме этого он в своем составе имеет центральный процессор (ЦП) с памятью 512 Кбайт для реализации операций с обширной логикой (например, агрегатных функций типа min, max и т. п.). Для облегчения входного (ВП) и выходного (ВЫП) потока данных РП содержит два адаптера иерархической памяти (АИП), а также входной модуль для подготовки кортежей отношений (например, перестановки значений атрибутов). Собственно операция реляционной алгебры реализуется в РП. Процессор слияния (ПСЛ) сортированных сегментов отношений предназначен для слияния сортированных сегментов отношений, а также в нем реализуются операции естественного соединения двух отношений и селекции отношения. Двенадцать процессоров сортировки (ПСО) предназначены для реализации конвейерной однопроводовой сортировки сегмента отношения объемом 64 Кбайт. ПСО и ПСЛ реализованы полностью аппаратно.

Иерархическая память в DELTA является наиболее сложной подсистемой, в функции которой входят:

управление СБП и УМП;

стадирование данных (в виде сегментов отношений) из УМП в СБП в соответствии с заявками РП;

селекция и вертикальная фильтрация отношений при помещении их в СБП с привлечением специального (атрибутного) метода хранения отношений в УМП;

поддержка индексных структур, кластеризация отношений в УМП и организация с их помощью быстрого поиска в УМП.

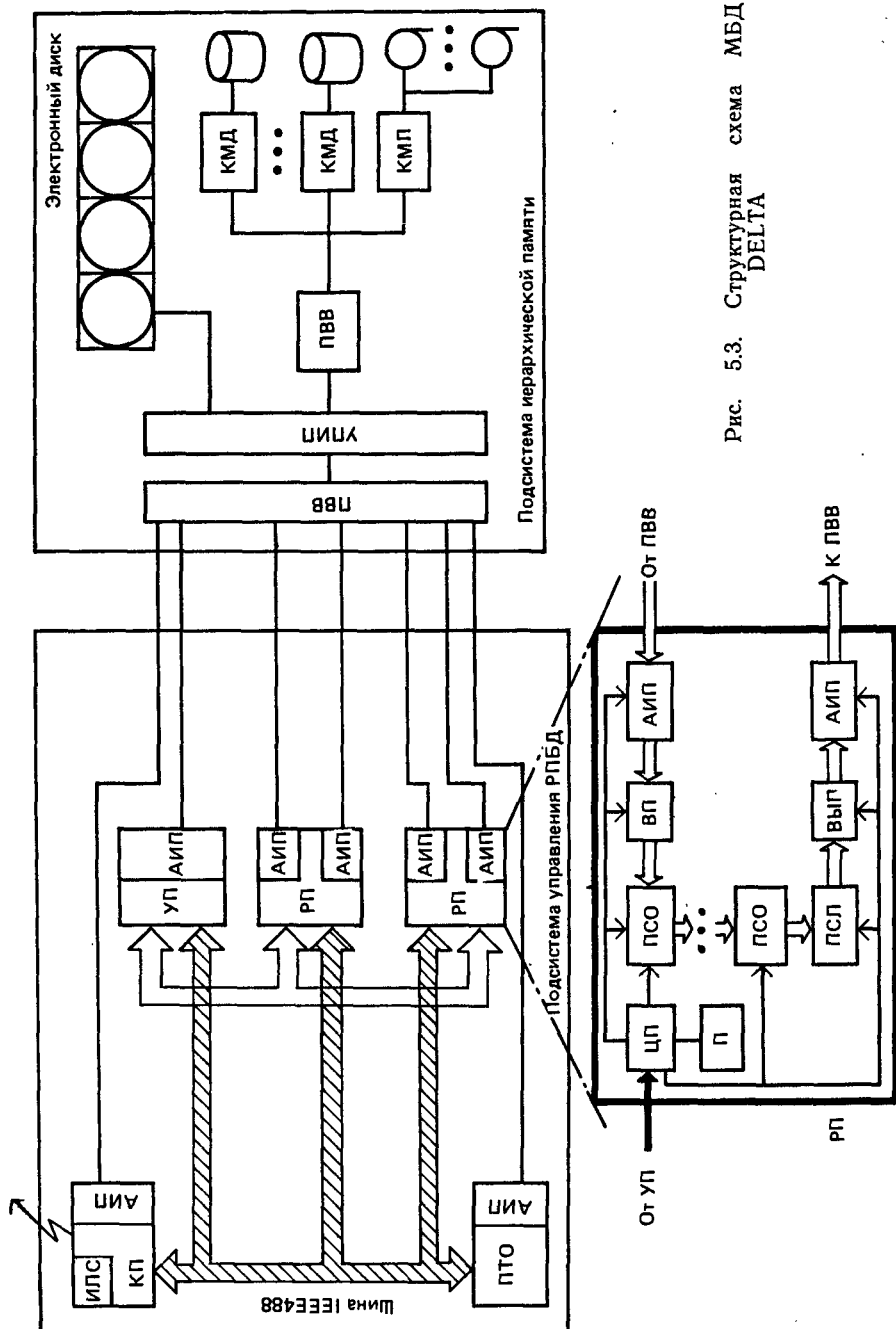


Рис. 5.3. Структурная схема МБД DELTA

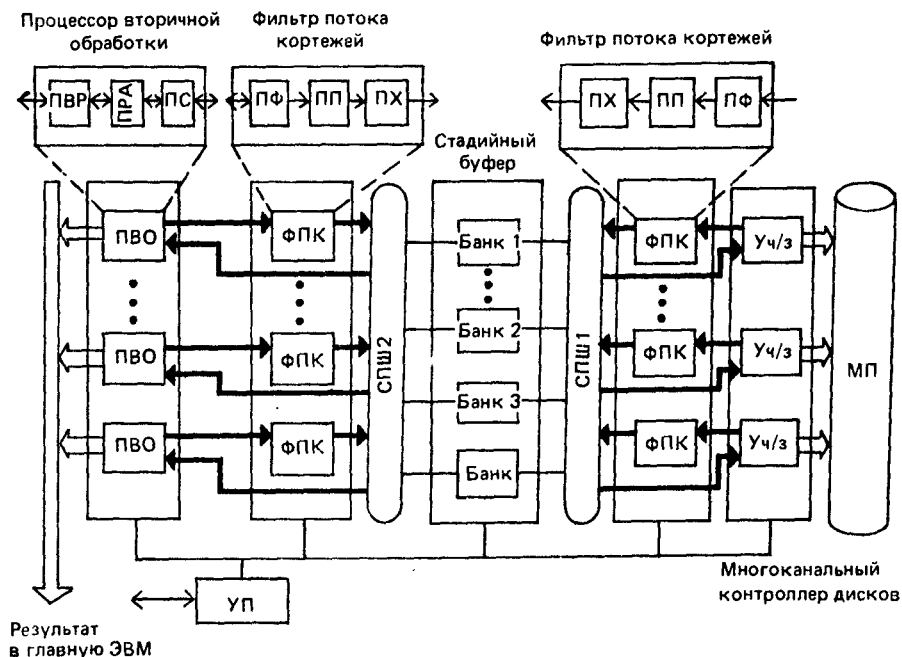


Рис. 5.4. Структурная схема МБД GRACE

Вторым примером МН МБД является также японский проект GRACE [Fushimi, 1985], структурная схема которого приведена на рис. 5.4. СБП реализована здесь набором электронных дисков на цилиндрических магнитных доменах. В качестве УМП использованы многоканальные НМД, в каждый канал которых встроены, кроме устройств чтения-записи (Уч/з), процессоры первичной обработки, названные фильтрами потока кортежей (ФПК). Каждый ФПК содержит:

процессор фильтрации (ПФ), осуществляющий в пределах дорожки МД собственно псевдоассоциативный поиск кортежей, удовлетворяющих заданному условию;

процессор проекции (ПП) и преобразования кортежей;

процессор хэширования (ПХ), реализующий динамическую сегментацию кортежей читаемого отношения.

Фильтр потока кортежей работает в конвейерном режиме и позволяет обрабатывать поступающие из УМП кортежи со скоростью их чтения (обработка «в полете»).

На уровне вторичной обработки применяются процессоры вторичной обработки (ПВО) и ФПК. Назначение ФПК — выполнять описанную выше обработку кортежей результирующих отношений, поступающих из ПВО в СБП. ПВО содержит наряду с процессором реляционной алгебры (ПРА), реализованным на основе универсального микропроцессора со своей локальной памятью, также аппаратный процессор сортировки отношений (ПСО) и процессор выдачи результата (ПВР) в канал главной ЭВМ. ПСО осуществляет потоковую сортировку сегмента отношения, поступающего из банка СБП в процессор реляционной алгебры. Двухпортовые банки СБП подсоединяются к процессорам обработки обоих уровней посредством специальных петлевых шин

(СПШ). Эти многоканальные шины с разделением времени осуществляют на каждом уровне обработки коммутацию любого процессора обработки к любому банку памяти и одновременную обработку нескольких банков памяти.

Отличительными особенностями данного проекта являются следующие структурные решения.

1. Метод опережающей подкачки кортежей из УМП в СБП сочетается здесь не только с первичной фильтрацией, но и со специальным распределением кортежей по банкам СБП. Эта так называемая динамическая хэш-сегментация позволяет выполнять операции реляционной алгебры на уровне вторичной обработки параллельно и без обменов несколькими ПРА, так что каждый ПРА реализует бинарную операцию над парой соответствующих сегментов отношений-операндов. Это является одним из источников повышения производительности МБД при выполнении операций реляционной алгебры.

2. Включение в цикл вторичной обработки фильтров потока кортежей, используемых в цикле первичной обработки, позволяет обрабатывать промежуточные отношения в СБП, так же как и исходные отношения БД, единообразно интерпретировать последовательность операций реляционной алгебры. Таким образом, выполнение транзакции, соответствующей сложному запросу к реляционной БД, заключается в многократном выполнении циклов первичной и вторичной обработки.

3. Предварительная и параллельная фильтрация данных со скоростью их поступления из УМП позволяет снизить объем перемещаемых из УМП в СБП данных, что является существенным источником повышения производительности МБД в целом. Этот механизм используется во многих (если не во всех) проектах МН МБД и считается признанным решением.

Как показали многочисленные исследования, СУБД не может быть эффективной, если большая часть ее работает под управлением операционной системы общего назначения. Поэтому повышение эффективности МБД связано с полной изоляцией СУБД в рамках МБД, т. е. реализацией функционально полных МБД, выполняющих все функции управления транзакциями. Учитывая сложность соответствующей операционной системы МБД, реализовать функционально полную и высокопараллельную МН МБД сложно.

Вторая основная проблема в создании высокопараллельных МН МБД, названная [DeWitt, 1983] «дисковым парадоксом», заключается в том, что скорость ввода-вывода современных УМП (одноканальные и многоканальные НМД с перемещающимися головками) является узким местом и ограничивает достижение высокого параллелизма в обработке. В МН МБД для решения этой проблемы в качестве кэш-диска применяется большая полупроводниковая буферная память.

Для решения этой проблемы некоторые авторы предлагают сетевые МБД, в которых распределенное хранение больших БД осуществляется на большом количестве НМД.

Сетевые МБД (см. рис. 5.1,б) воплощают принципы однородности структуры, сегментации данных в устройствах массовой памяти и распределении процессоров обработки по УМП. Таким образом, основная идея сетевых МБД [Рыбкин, 1980] — приближение дешевой обрабатывающей логики (в виде универсальных микропроцессоров) к УМП и связывание таких «обрабатывающих хранилищ» в сеть. Учитывая быстро снижающуюся стоимость процессоров обработки и жестких НМД и успехи в технике коммуникации процессоров, в составе такой сети может быть сотни УМП, с каждым из которых соединен свой обрабатывающий процессор. Примерами таких проектов являются MBDS [Hsiao, 1983]; GAMMA [DeWitt, 1986]. В перспективе развития сетевых МБД некоторые авторы видят создание МБД на основе вычислительной систолической среды. Так, проект NODD [Bis, 1985], схема которого изображена на рис. 5.5, реализован в виде регулярной решетки, в узлах которой размещены процессорные элементы (ПЭ). С каждым ПЭ связаны своя локальная память (ЛП) и устройство массовой памяти (УМП) в виде жесткого НМД.

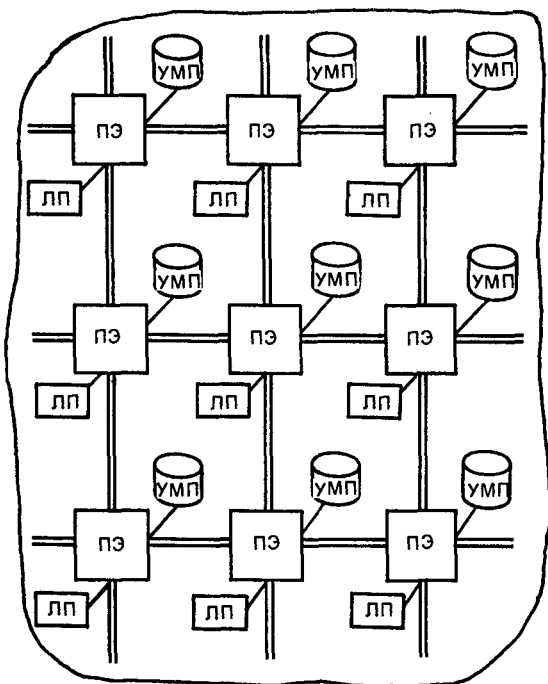


Рис. 5.5. Структура систологической МБД NODD

Предполагается также замена УМП энергонезависимой полупроводниковой памятью соответствующей емкости («силиконовая» систолическая МБД). Назначение ЛП в каждом узле — хранение программ обработки, копии управляющей программы, буфера для обмена сообщениями и кэш-памяти для своего УМП (3% каждого локального УМП находится в этой локальной кэш-памяти). Для целей надежности пространство на каждом УМП разделено на части: одна часть для хранения части БД, принадлежащей своему узлу, другая — дублирует данные четырех соседних узлов. Особенностью является и то, что управление выполнением транзакций в таких сетевых МБД полностью распределено, так что каждый процессор может взять на себя роль управляющего.

Особый интерес приобретает создание систолических МБД в связи с появлением серийных однокристалльных транспьютеров, содержащих наряду с процессором и памятью каналы (порты ввода-вывода). Например, промышленный транспьютер фирмы INMOS IMS T414 имеет следующие характеристики [Электроника, 1986]. В одном кристалле реализован 32-разрядный процессор быстродействием до 10 млн. опер./с, статическое ОЗУ на 2 Кбайт, четыре канала связи, 32-разрядный интерфейс памяти и контроллер динамического ОЗУ. Конструктивно транспьютерная матрица, являющаяся основным элементом систолических транспьютерных МБД (см. рис. 5.5), может быть реализована посредством серийных транспьютерных плат IMS B0003 той же фирмы. Эта двойная европлата содержит четыре транспьютера T414, связанных между собой портами связи, четыре устройства динамической памяти по 256 Кбайт каждое и четыре внешних порта ввода-вывода. Возможно, в ближайшее время применение таких транспьютерных плат переведет проекты систолических МБД из области теоретических исследований в область практической реализации.

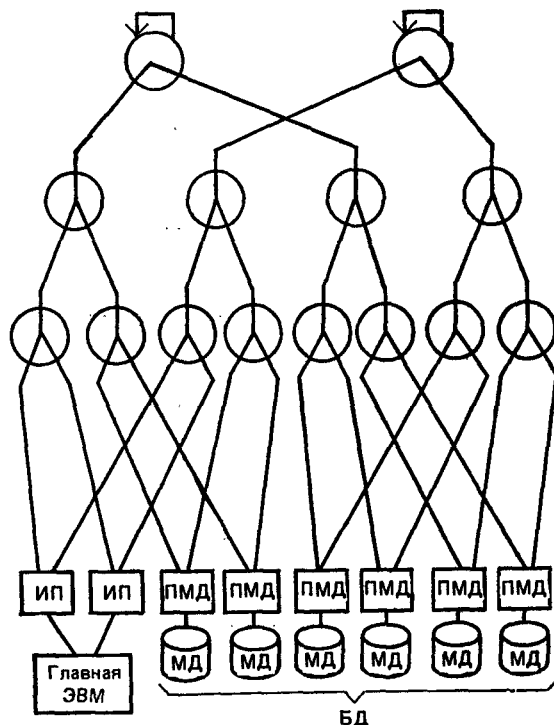


Рис. 5.6. Сетевая МБД Teradata DBC 1012

Основной проблемой в распределенных (сетевых) МБД является оптимальная кластеризация данных по локальным УМП и поддержка соответствующей распределенной индексации. В GAMMA, например, предлагается кластеризация каждого отношения по всем УМП (в соответствии с хешированием значений ключевых атрибутов и созданием распределенного по УМП индекса этих значений). В NODD предлагается равномерное распределение отношений по узлам решетки. Между конкретными кортежами разных отношений, для которых действуют семантические связи, существуют указатели, задающие расположение связанных кортежей (номера узлов и их адреса в УМП). Таким образом, запрос в БД возбуждает связи между кортежами в узлах решетки и порождает поток данных между ними. Это позволяет реализовать в такой МБД потоковую обработку сложных запросов на основе модели «активного графа» [Вис, 1985].

К классу сетевых относится коммерческая МБД фирмы Teradata DBC 1012, которая интенсивно распространяется и находит широкое применение в различных информационных системах [DBM, 1985]. На рис. 5.6 изображена конфигурация DBC 1012 с восемью обрабатывающими процессорами ПМД (на базе i80386), каждый из которых имеет НМД и подключается к коммуникационной сети типа двоичного дерева (Y-сеть). В узлы этой сети встроены сетевые высокоскоростные процессоры и программируемые логические матрицы, реализующие функции управления сетью. Y-сеть позволяет осуществлять дуплексные обмены между обрабатывающими процессорами. В эту же сеть подключаются

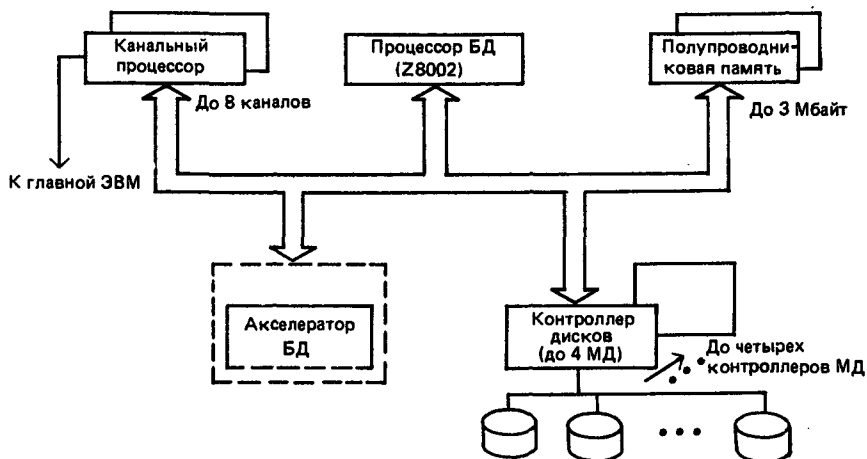


Рис. 5.7. Структурная схема МБД IDM 500

коммуникационные процессоры (ИП) для осуществления интерфейса с главной ЭВМ. Каждый обрабатывающий процессор обеспечивает поддержку всех операций реляционной алгебры, достаточных для выполнения операторов SQL, поддержку своей части БД, а также выполнение всех функций управления транзакциями над своей частью БД, в том числе защиту целостности, восстановления и т. д. Образцы DBC 1012 включают до 128 процессоров и имеют распределенную по обрабатывающим процессорам полупроводниковую память емкостью 412 Мбайт на один процессор. Общая емкость массовой памяти составляет до 1000 Гбайт и общее быстродействие — до  $10^9$  опер./с.

Два свойства DBC 1012 характерны для всех сетевых МБД:

обеспечение возможности увеличения мощности наращиванием числа обрабатывающих процессоров, так что производительность при этом растет линейно (показатель линейности роста производительности DBC 1012 от числа процессоров составляет 97%);

обеспечение надежности функционирования за счет дублирования данных в локальных УМП (т. е. обеспечивается работа без краха системы при выходе из строя отдельных процессоров или УМП).

Третье направление исследований в области МБД заключается в создании недорогих коммерческих устройств на серийных процессорных элементах с шинным интерфейсом (топология таких МБД изображена на рис. 5.2,а). В качестве примера рассмотрим МБД фирмы Britton Lee IDM 500, структурная схема которой изображена на рис. 5.7. Хотя эти изделия не ориентированы на высокопараллельную обработку и содержат ограниченное число функциональных процессоров, они удовлетворяют сформулированным выше принципам МН МБД и полностью реализуют все основные функции МБД. Структурная схема коммерческих МБД является частным случаем МН МБД (см. рис. 5.1,а). Роль СВП выполняет полупроводниковая память, к которой через общую шину подключаются периферийные контроллеры НМД со встроенными микропроцессорами AMD 2901, процессор обработки (процессор БД) на основе Z8002 и до 8 канальных процессоров для подключения к главной ЭВМ (канал IBM 370, интерфейс с VAX 750) или подключения к локальной сети (Ethernet). Кроме того, к общей шине может подключаться особый функциональный процессор (акселератор БД) для выполнения тех операций, которые являются узким ме-

стом (например, сортировка отношений). Старшая модель IDM 500/XL с емкостью внешней памяти более 1 Гбайт на жестких МД и 500 Мбайт на МЛ имеет производительность 1000 транзакций/мин и одновременно обслуживает до 400 пользователей.

Развитием этого направления в разработках фирмы Britton Lee является реляционный файловый сервер (data/file server) RS310 — автономное устройство, подключаемое к локальной сети Ethernet или непосредственно к главной ЭВМ по интерфейсу RS232. Он включает:

- собственно процессор базы данных (1 плата) на основе Z8000 (10 МГц);
- соединенную с этим процессором оперативную память емкостью 1 Мбайт на одной плате;

- два жестких диска типа винчестер (5 1/4 дюйма) по 80 Мбайт каждый с соответствующим контроллером;

- контроллер кассетной МЛ с 60 Мбайт на кассете (Streaming tape 1/4 дюйма);

- до четырех интерфейсных плат двух типов (интерфейс RS232 с восемью выходами или интерфейс локальной сети Ethernet).

Каждая интерфейсная плата содержит процессор Z8000 и свою локальную память. RS310 может быть использован или как автономная СУБД с выходным языком SQL, поддерживая при этом все функции СУБД, за исключением первого этапа трансляции с SQL (управление транзакциями, параллельное выполнение запросов, откаты и восстановления, автоматическую оптимизацию запросов и т. п.), или как интегрированная система управления файлами. При этом RS310 выступает для главной ЭВМ в качестве интеллектуального контроллера с буферизацией и удовлетворяет интерфейсу SCSI (Small Computer System Interface). RS310 обеспечивает одновременную работу до 50 пользователей и выполняет одновременно до 10 запросов. Ближайшая перспектива развития RS310 — увеличение внешней памяти до восьми НМД емкостью 478 Мбайт и МЛ емкостью 300 Мбайт.

Дальнейшим развитием такого подхода к созданию коммерческих МБД является их реализация на модульной параллельной мультимикропроцессорной системе типа систем S27 и S81 фирмы Sequent и систем серии 9000 (9810, 9820) фирмы Pyramid-Sybase. На рис. 5.8 изображена структурная схема нового изделия фирмы Pyramid — система 9810(9820), являющаяся специализированной ЭВМ для БД. Эта специализированная машина предназначена для автономной поддержки СУБД Sybase с входным языком SQL, а также для поддержки прикладных информационных систем на основе этой СУБД для автоматизации конторской деятельности, разработки программного обеспечения и т. п. Система работает как data computer в сети ЭВМ и имеет интерфейс не только с локальной сетью Ethernet, но и X25, telenet, datra. Общая дисковая память достигает 15 Гбайт. Основная память, подключаемая к устройству управления памятью в виде плат до 4 и 16 Мбайт, может наращиваться до 128 Мбайт. В системе поддерживается виртуальное адресное пространство 4 Гбайт со страницами в 2048 байт. В качестве процессоров обработки выступают один или два спецпроцессора (CPU), реализованные в виде 32-разрядных процессоров с RISC-архитектурой (см. § 4.2). CPU имеет следующие характеристики:

- время цикла — 100 нс;
- число 32-разрядных регистров — 528;
- кэш-память инструкции — 16 Кбайт;
- кэш-память — 64 Кбайт.

В RISC-процессорах реализован конвейерный режим выполнения инструкций.

Основой системы является собственная сверхбыстрая шина xtpend (40 Мбайт/с), работающая по принципу коммутации сообщений. Интеллектуальный процессор ввода-вывода (ПВВ) реализован на базе микропроцессора AMD 29116 с быстродействием 5 млн. опер./с и содержит 14 параллельных



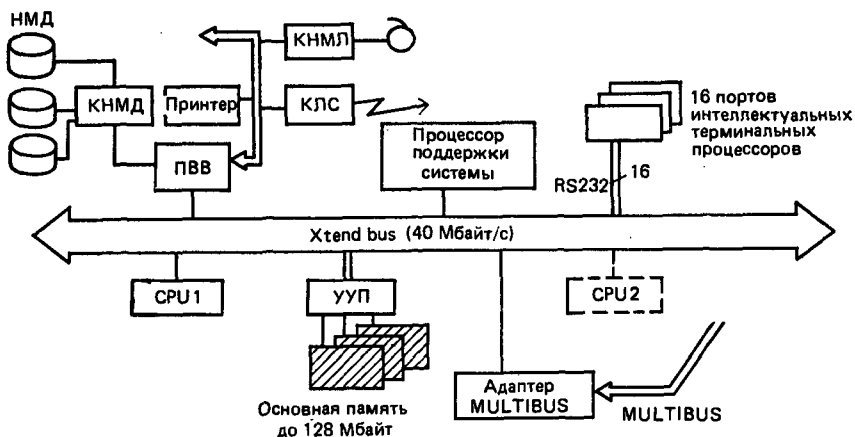


Рис. 5.8. Специализированная машина для БД PYRAMID S 9810 (9820)

DMA-контроллеров, общая пропускная способность которых 11 Мбайт/с. ПВВ обслуживает периферийные устройства, контроллеры НМД (скорость передачи в которых до 2,5 Мбайт/с) и контроллеры локальной сети (КЛС).

К общей шине подключается до 16 портов с интерфейсом RS232 для обслуживания интеллектуальных терминальных процессоров, с помощью которых к системе могут подключаться терминальные пользователи. Подключение к шине адаптера шины MULTIBUS открывает широкие возможности для подключения вспомогательных внешних устройств, в которых реализован интерфейс этой шины.

Управление системой осуществляется процессором поддержки системы, в функции которого входят также диагностика всех устройств и системы в целом, сервисная служба системы и т. п. В этом процессоре функционирует так называемая двухпортовая многопроцессорная операционная система, которая соединяет в себе две версии UNIX ОС: System V. AT&T и 4.0 Berkly.

### Перспективы развития МБД

Создание высокопроизводительных МБД связывается с решением следующих проблем, по которым ведутся интенсивные исследования.

1. Создание специализированных архитектур МБД, сочетающих достоинства горизонтального параллелизма при выполнении одной операции с функциональным параллелизмом при выполнении последовательности операций и транзакций. Особую роль здесь играет реализация конвейерной потоковой обработки (data flow) применительно к операциям реляционной алгебры.

Управление выполнением запросов при этом должно происходить собственными потоками данных, что облегчает задачу программного контроля выполнения операций, синхронизации их и т. п. Например, в проекте DFRU [Glinz, 1983] предпринята попытка аппаратно реализовать потоковую обработку в регулярной структуре обрабатывающих процессоров (рис. 5.9). Основным обрабатывающим элементом является универсальный компаратор кортежей (К). В матрице компараторов кортежей, соединяемых коммутирующими сетями, возможна динамическая коммутация выходов процессоров обработки  $i$ -го уровня со входами процессоров  $(i+1)$ -го уровня. В каждой строке матрицы по одному арифметическому процессору (АП) для реализации агрегатных функций. Связи между процессорами устанавливаются в соответствии с дугами дерева запроса (дерева реляционных операций). Это позволяет отображать де-

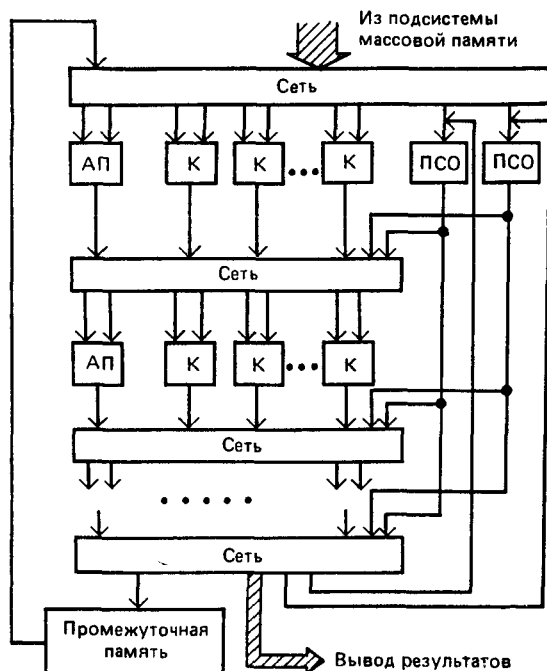


Рис. 5.9. Структура подсистемы потоковой обработки в МБД DBFRU

рево запроса в дерево процессоров, вырезаемых в данной матрице, так что каждой операции назначается один процессор обработки. После этого исходные отношения в потоке читаются из подсистемы массовой памяти и обрабатываются конвейерно в настроенном дереве процессоров, так что кортежи, образованные в операции  $i$ -го уровня, через коммутирующую сеть попадают в соответствующий процессор  $(i+1)$ -го уровня. На рис. 5.10 проиллюстрирован этот процесс для следующего запроса: «Выдать имена поставщиков, которые поставляют более 100 деталей типа 1» применительно к трем исходным отношениям:

- (R1) ПОСТАВЩИК (КОД ПОСТАВЩИКА, имя);
- (R2) ДЕТАЛЬ (КОД-ДЕТАЛИ, ТИП-ДЕТАЛИ, НАИМЕНОВАНИЕ);
- (R3) ПОСТАВКА (КОД-ДЕТАЛИ, КОД-ПОСТАВЩИКА, КОЛИЧЕСТВО).

Появление на входе компаратора кортежа исходного отношения для операции селекции (select) или двух исходных кортежей для операции соединения (join) активизирует его, и после выполнения операции и выдачи результирующего кортежа он переходит в состояние ожидания.

Процессоры сортировки отношений подключаются перед бинарными операциями или перед операцией удаления дублей. Промежуточная память используется для закливания потока кортежей, если длина дерева запроса больше числа строк в обрабатываемой матрице, или для передачи промежуточных отношений между двумя деревьями запросов.

В матрице процессоров возможно одновременное выполнение нескольких запросов, каждый из которых отображен в свое дерево процессоров.

Необходимость в сортировке объясняется тем, что реализация бинарных операций реляционной алгебры (с нелинейной сложностью  $O(n^2)$ , где  $n$  — кар-

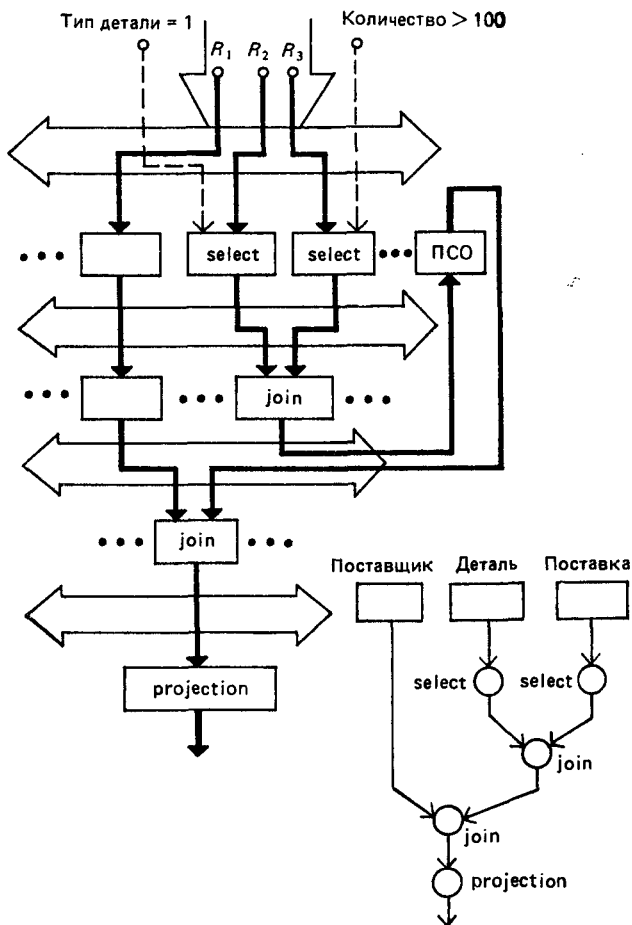


Рис. 5.10. Реализация дерева операций в матрице процессоров

динальность отношений) в потоковом режиме, когда единицей обмена между операциями являются кортежи или страницы отношений, возможна, только если отношения одинаково упорядочены. Поэтому операция сортировки в конвейерном и потоковом режимах обработки является узким местом, и требуется ее аппаратная реализация, удовлетворяющая следующим требованиям:

высокая скорость и близкая к линейной зависимость времени сортировки от объема отношения;

один проход при реализации сортировки;

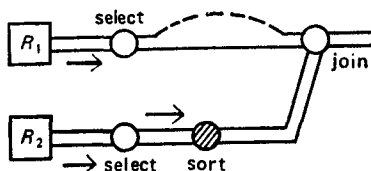
конвейерный режим обработки потока данных;

наличие внутренних буферов объемом не менее страницы отношения (64, 128, 256 Кбайт);

возможность исключения дубликатов кортежей при сортировке отношения.

В исследовательских проектах МБД Delta, RDBM, DSDBC, PPRQP используются также потоковые процессоры внутренней сортировки страниц отношений.

Рис. 5.11. Влияние задержки при сортировке на потоковый режим обработки кортежей



Необходимо отметить, что даже применение аппаратных потоковых сортировщиков не решает полностью проблему потокового выполнения бинарных операций. Сортировщик может начать выдачу кортежей сортированного отношения только после получения на входе последнего кортежа исходного отношения, да и то с задержкой. Например, процессор сортировки в Delta имеет задержку  $(2lm + N - 1)t$ , где  $l$  — длина кортежа в байтах,  $t = 330$  нс — время пересылки байта между сортирующими элементами,  $m$  — число сортируемых кортежей,  $N$  — число сортирующих элементов. Таким образом, наличие даже аппаратной потоковой сортировки прерывает и замедляет общий поток кортежей между реляционными операциями. А любое замедление входного потока одного отношения-операнда требует для другого операнда бинарной операции наличия промежуточного буфера. Поэтому конвейер кортежей при потоковой обработке должен иметь вид «растягиваемых рукавов» (рис. 5.11). Все это может свести к минимуму преимущества потоковой обработки.

2. Создание ассоциативной памяти большой емкости для системного буфера МБД. Как известно [Ozkaragachan, 1986], использование ассоциативной памяти в качестве СБП позволяет существенно повысить эффективность поисковых и некоторых других операций в МБД на уровне вторичной обработки. Интерес представляет гибридная ассоциативная память, которая имеет один порт — обычный для подключения памяти к общей шине, а второй — ассоциативный для подключения к соответствующему контроллеру.

Примером такой памяти является изделие фирмы Сименс [DeWitt, 1985], структурная схема подключения которого к общей шине представлена на рис. 5.12,а. Наличие порта обычной адресации позволяет осуществлять подкачку данных в ассоциативную память через общую шину из УМП. Ассоциативная память в этом изделии содержит 64 параллельные линии обработки (шириной 8 разрядов каждая), так что информация из памяти поступает в каждый из 64 процессорных элементов побайтно (в виде столбца байтов длиной 256 К). Длина обрабатываемого слова 256К, общая емкость ассоциативной памяти 16 Мбайт. Модуль памяти, обрабатываемый каждым процессорным элементом, 256 Кбайт. Каждый модуль памяти имеет адрес на общей шине, и информация в него может записываться автономно по первому порту. Кроме того, внутри столбца — своя адресация в пределах 256К, доступная контроллеру ассоциативной памяти. Ассоциативный поиск осуществляется синхронно всеми (или частью) процессорными элементами.

Структура каждого процессорного элемента изображена на рис. 5.12,б. Каждый процессорный элемент содержит АЛУ и тест-устройство, два входных регистра А и В, регистр маски М и блок регистров С. Все эти устройства выходят на внутреннюю шину памяти (М-шину). Процессорный элемент и модуль памяти реализованы двумя кристаллами. Общий контроллер управляет синхронной работой всех процессорных элементов, воспринимает результат и при поиске постоянно фиксирует текущий адрес в пределах 256К, с которым в данный момент работают процессорные элементы. Факт нахождения релевантной информации в каждом процессорном элементе фиксируется в контроллере для последующего извлечения информации. В такой ассоциативной памяти наиболее эффективно реализуется операция поиска вхождений по заданному образцу. Особый интерес представляет использование такой памяти в машинах баз знаний, где операция поиска вхождений по образцу является основной.

3. Использование процессора потоковой сортировки отношений для слияния отношений. Например, в проекте Delta он предназначен для потокового

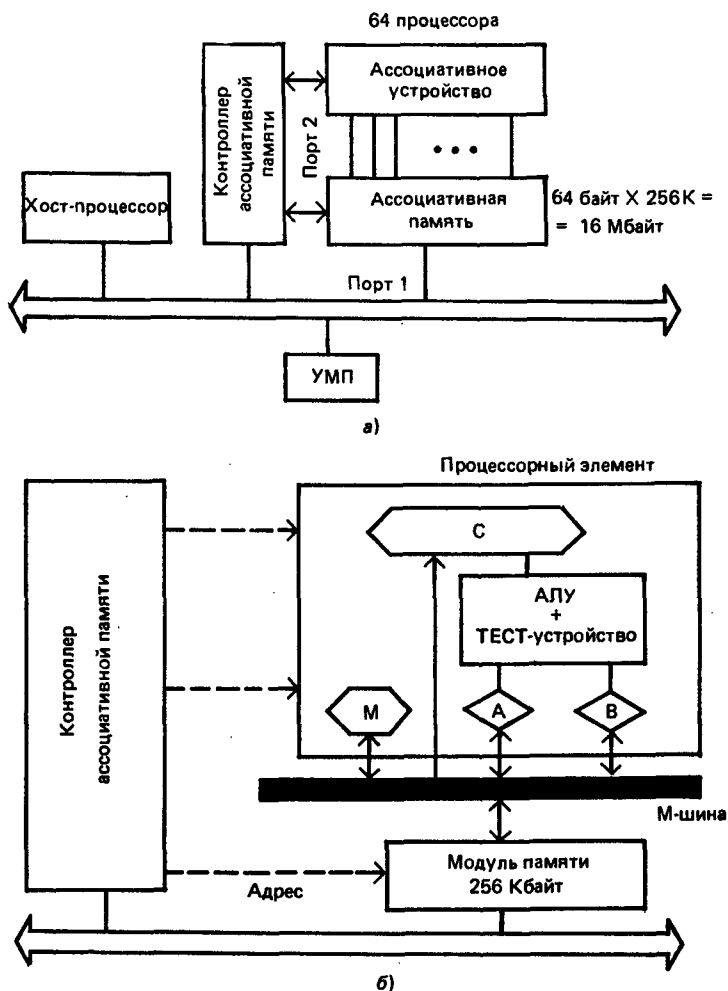


Рис. 5.12. Гибридная двухпортовая ассоциативная память:  
а — общая схема; б — схема процессорного элемента

слияния двух сортированных страниц отношения со скоростью поступления кортежей второй страницы. На базе такого двухвходового процессора слияния может быть реализовано устройство соединения двух отношений (Delta).

Во многих проектах предлагается аппаратная реализация операций реляционной алгебры на основе универсальных компараторных процессоров (компараторов). Пример такого компаратора (проект DFRU) представлен на рис. 5.13. Компаратор (К) предназначен для реализации операций селекции, проекции, сужения и соединения отношений и используется в процессорной матрице (рис. 5.9). Он имеет два буфера объемом, достаточным для помещения одного кортежа, и управляется потоком кортежей, поступающих по двум входным ли-

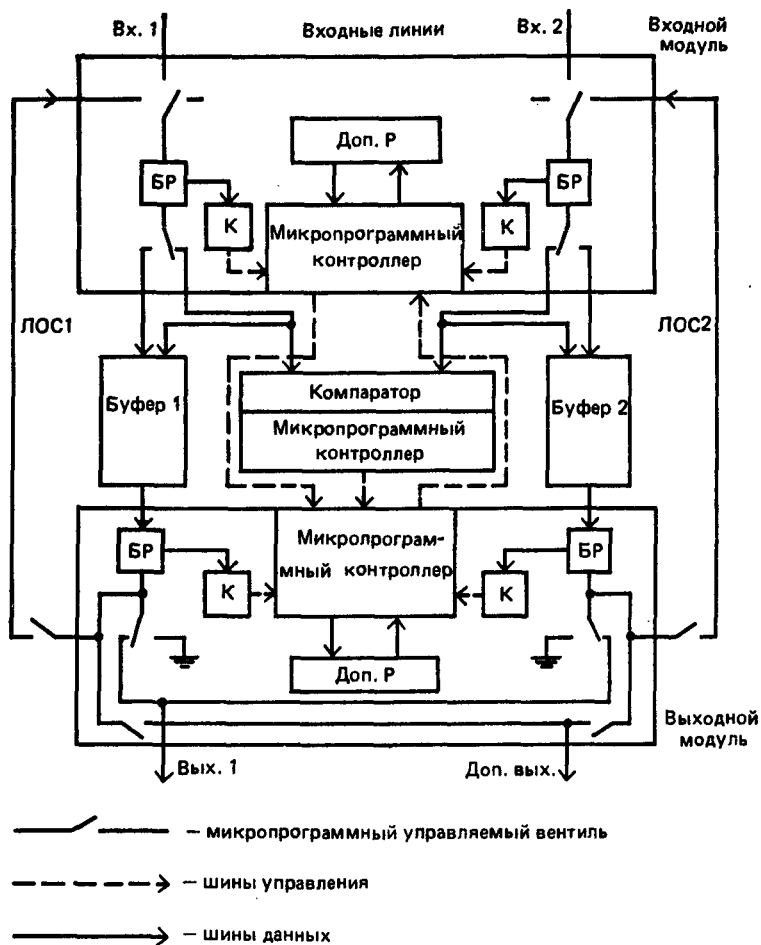


Рис. 5.13. Универсальный компараторный процессор для реляционных операций: БР—8-разрядный буферный регистр; Доп. Р — дополнительный регистр; К — компаратор; ЛОС — линии обратной связи

ниям побайтно. Компаратор может находиться в разных состояниях, управляемых тремя внутренними линиями. Он управляется микропрограммой, запускаемой входными коротежами. Режим работы микропрограммы определяется текущим состоянием компаратора. Компаратор содержит три микропрограммных процессора (модуля): ввода, сравнения и выхода, и набор внутренних регистров (Доп. Р, БР). Входной модуль читает коротежи на своих входных линиях и направляет каждый коротеж побайтно в буфер или/и в компаратор (в зависимости от своего внутреннего состояния). Дополнительные компараторы необходимы для анализа управляющих битов (флагов), таких как признак окончания отношения, который оканчивает обработку. В центральном компараторе происходит сравнение значений заданных атрибутов из входных короте-

жей (побайтно). В компаратор поступает не весь кортеж, а только сравниваемые значения. Этим управляет входной модуль по своей микропрограмме, синхронизируясь по заданному расположению этих сравниваемых значений. Весь кортеж находится в буфере, и после сравнения заданных атрибутов компаратор задает выходному модулю команду на вывод кортежей из соответствующих буферов. Вывод может осуществляться:

на выходную шину, если сравнение было успешным;

в линию обратной связи (ЛОС), если, например, из входного модуля по шине управления пришла команда о том, что в следующем кортеже второго отношения то же значение атрибута соединения (для операции соединения) и этот выводимый кортеж надо повторно с ним сравнить. При этом соответствующая входная линия перекрывается и замыкается на ЛОС, пока на входе не появится кортеж второго отношения с новым значением атрибута соединения;

удаление текущего кортежа из выходной последовательности при неуспешном сравнении (выходная линия замыкается на «землю»). Для операции соединения в выходном модуле может выполняться сцепление выходных кортежей. Режим потоковой обработки здесь обеспечивается тем, что микропрограммные процессоры (входной, компараторный и выходной) управляются входными последовательностями кортежей через изменение внутренних состояний. При этом определенная микропрограмма выполняется, если модуль находится в соответствующем состоянии, зависящем от поступления входных кортежей.

При выполнении операции селекции в Буфер 2 загружается «псевдокортеж», соответствующий заданному условию поиска. (Очевидно, таким образом можно реализовать селекцию по условию, не содержащему дизъюнкций.) Кортежи отношения, поступающие на Вх.1, пропускаются через компаратор, а «псевдокортеж» циркулирует по ЛОС2, Буфер 2 и компаратору. Входной модуль синхронизирует этот процесс. При успешном сравнении в компараторе кортеж отношения посылается на Вых.1, иначе на «землю». Аналогично рассмотрению реализован реляционный процессор РП (см. рис. 5.3) в МБД Delta, но с большим разнообразием выполняемых функций.

4. Аппаратная реализация потоковой фильтрации данных непосредственно в каналах УМП. Такая фильтрация позволяет снизить объемы данных, передаваемых из массовой памяти на обработку, что является существенным источником повышения производительности МБД в целом. Потоковые процессоры фильтрации (фильтры) должны удовлетворять следующим требованиям:

скорость обработки должна соответствовать скорости чтения НМД, чтобы избежать холостых оборотов МД;

необходимость использования двух коммутируемых буферов объемом в одну дорожку МД для обеспечения непрерывности чтения;

возможность пропускать в выходной канал только релевантные поисковому условию данные (горизонтальная фильтрация);

возможность формировать на выходе часть входной записи в соответствии с заданным условием (вертикальная фильтрация);

задание поисковых условий в виде дизъюнктивной нормальной формы элементарных поисковых условий или в виде поисковых образов;

объем поисковых условий должен определяться допустимой скоростью обработки.

Достаточно полный обзор существующих процессоров фильтрации см. в [Ozkarahan, 1986].

5. Устранение препятствия в увеличении производительности многопроцессорной МБД с двумя уровнями обработки и системной буферной памятью за счет улучшения системы коммутации и связи процессоров обработки с СБП и между собой. Это определяется интенсивностью обмена данными между процессорами и СБП и объемом этих данных, которые, как правило, являются частями файлов (отношений БД). Кроме этого организация параллельной работы процессоров требует интенсивного обмена сообщениями между процессорами. Ориентировочные требования к системе коммутации [Тапака, 1984] в МБД с высокой степенью параллельности:

скорость передачи данных — 10—80 Мбайт/с;  
число подключаемых автономных банков памяти — 100;  
число подключаемых процессоров обработки — 100;  
отсутствие конфликтов.

В этой же работе дается ряд перспективных методов реализации высокоскоростных сетей коммутации (до  $10^3$  входов) и реализации на их основе многопортовой системой буферной памяти для МБД.

Фильтры представляют собой технические или программные средства для выделения данных, удовлетворяющих каким-либо условиям. В реляционной модели данных выбор по условию соответствует операциям проекции и селекции реляционной алгебры. В задачах обработки текста требуется, как правило, выбрать текст, содержащий данное слово. Те же задачи выбора данных по условию могут решаться и с помощью использования циклов или хэширования. Фильтры, однако, позволяют проверять более сложные условия, чем «равно» (для хэширования возможна только такая проверка), «не равно», «больше», «меньше». Фильтры не требуют предварительной структуризации данных. Вместе с тем использование фильтров предполагает полный просмотр отношения (или, если в базе хранятся инвертированные файлы, полный просмотр значений атрибутов, участвующих в условии). Фильтры могут применяться и в комбинации с индексированием и хэшированием [Holaar, 1985].

Общая схема использования фильтров следующая. Процессор, анализирующий запрос, выделяет условие селекции. Выделенное условие передается в специальный процессор (или группу процессоров), который является программируемым фильтром. Этот процессор может быть как специализированным, так и процессором общего назначения.

Затем фильтр читает последовательно кортежи отношения, проверяя на них заданное условие и отбирая удовлетворяющие этому условию кортежи. Отобранные (релевантные) кортежи передаются другим процессорам для дальнейшей обработки. Обработка может осуществляться параллельно с работой фильтра.

В ряде проектов МБД используются фильтры, которые моделируют для проверки условий конечные автоматы. При такой реализации скорость проверки условия практически не зависит от его сложности (если размер памяти фильтра достаточен для моделирования конечного автомата, проверяющего условие).

Бинарные операции (объединение, пересечение, соединение) над отсортированными отношениями также могут быть реализованы с помощью фильтров [Garmetman, 1983].

Специализированный фильтр работает приблизительно в два раза быстрее, чем происходит загрузка входного буфера с диска. В процессе загрузки фильтр может читать уже загруженную часть памяти. Кэш-память служит промежуточной для обмена между дисковой памятью и буферами.

Когда в качестве фильтрующего процессора используется процессор общего назначения, входной буфер разбивается на два. Контроллер загружает блоки входного файла поочередно в один из двух буферов. Загрузка ведется непосредственно с диска, так как фильтрация идет приблизительно в три раза медленнее чтения. Кэш-память в этом случае не нужна. Результат фильтрации записывается в один из двух выходных буферов. По заполнении буфера его содержимое сбрасывается на диск.

К недостаткам фильтров, моделирующих конечные автоматы, следует отнести высокие требования к размерам памяти фильтра [VERSO, 1986].

Другой способ проверки условий осуществляется прямым фильтром, структура которого соответствует структуре проверяемого условия. Основными блоками фильтра являются компаратор, управляющее устройство и логическое устройство. Компараторы проверяют истинность простых условий. Их число ограничено и фиксировано в аппаратуре. Современная технология СБИС позволяет строить прямые фильтры с числом компараторов порядка 100.

Управляющее устройство организует работу фильтра, логическое устройст-



во обрабатывает полученные значения и генерирует окончательное значение истинности сложного условия.

На вход компаратора подается запись вида

<атрибут, оператор, значение-операнда>

или

<атрибут, условие, атрибут>

где <условие> — одно из условий сравнения. Результатом работы компаратора является признак ИСТИНА или ЛОЖЬ.

Передаваемое в фильтр формализованное условие представлено в дизъюнктивной или конъюнктивной нормальной форме. Для определенности будем говорить далее о дизъюнктивной нормальной форме. В этом случае условие представляет собой дизъюнкцию мономов. Каждый моном есть конъюнкция простых условий. На каждом шаге компаратор проверяет очередное простое условие, и постепенно формируется окончательный результат проверки всего сложного условия.

С каждым компаратором соединен логический блок, который вычисляет очередное промежуточное значение окончательного результата по результату работы компаратора, предыдущему промежуточному значению и логической связке. На каждой плате размещен контроллер и набор соединенных компараторов и логических блоков.

Управляющее устройство выполняет следующие функции:

декодирование, вычисление длины операндов и подготовка данных к вычислениям;

определение концов операндов и установка соответствующих признаков;

управление входным и выходным буферами. Входной буфер не является необходимым ввиду возможности прямого доступа управляющего устройства к основной памяти.

Операнды проще всего размещать во внешней памяти, общей для всех компараторов. Время обращения к общей памяти налагает в этом случае ограничения на число компараторов в фильтре. Если время ввода одной записи 100 нс, а время работы компаратора 400 нс, то нет смысла иметь более четырех компараторов. Использование СБИС-технологии позволяет обеспечить каждому компаратору достаточную внутреннюю память (порядка 256 байт).

Существует ряд проблем, которые находятся пока за рамками исследования по МБД.

1. Повышение эффективности хранения данных в больших БД часто связывают со сжатием данных, специальным кодированием данных и т. п. Но псевдоассоциативный поиск и фильтрация данных непосредственно в УМП трудно реализуемы, если данные в УМП хранятся в сжатом и закодированном виде. Пока только в единственном проекте DS DBS предпринята попытка решить эту проблему.

2. При разработке МБД совсем не рассматривается проблема обеспечения интерактивного взаимодействия пользователя с БД посредством графического дисплея. Если терминалы работают под управлением МБД, то сложность ОС МБД существенно возрастает, что может привести к деградации общей производительности МБД. Если терминалы пользователя работают под управлением главной ЭВМ, то растет объем данных, передаваемых от МБД в главную ЭВМ и наоборот.

3. Повышение производительности МБД обычно связывается со скоростью выполнения операций, деревьев запросов, отдельных транзакций и смеси таких транзакций. При этом выдача данных терминальному пользователю начинает осуществляться только после выполнения последней реляционной операции в последовательности операций, соответствующих запросу. Иногда для принятия решения достаточно нескольких кортежей, являющихся результатом этой последовательности (дерева запроса). Увеличение реактивности МБД при выдаче этих нескольких кортежей, удовлетворяющих запросу, часто противоречит

увеличению традиционной пропускной способности МБД. Решить эту проблему можно только реализацией в МБД такого режима потоковой обработки отношений [Андон и др., 1983], при котором реляционная операция начнет выдавать результирующие кортежи, не ожидая появления целиком сформированных отношений-операндов. Для ряда операций реляционной алгебры сложности  $O(n^2)$  (где  $n$  — кардинальность отношений-операндов) реализация такого режима трудно разрешима, например для операции сортировки отношений.

4. Обеспечение целостности БД при параллельных обновлениях в МБД с высокой степенью внутреннего параллелизма, а также живучести таких систем и их надежного функционирования — также серьезная проблема.

5. Разработка единой методологии проектирования МБД исходя из заданного набора требований (объем и тип БД, типы и частота запросов, сфера применения и т. п.). В настоящее время проектирование МБД основано на интуитивных соображениях, и отсутствуют механизмы предварительной оценки производительности, такие как для параллельных систем вычислительного типа.

Наличие указанных проблем в проектировании МБД заставляет некоторых авторов [Hawthorn, 1981] на вопрос «Существует ли идеальная МБД?» ответить следующим образом: «Идеальная МБД, если она существует, должна быть, очевидно, слишком дорогостоящей и слишком сложной, чтобы ее можно было использовать универсально в каждой области применений».

Лучшей рекомендацией в данном случае является разработка семейства МБД, позволяющая осуществить необходимый выбор для каждого специфического применения. Например, мультипроцессорная машина псевдоассоциативного поиска в больших файлах библиографических систем или многопроцессорные МБД для поддержки реляционных операций как составная часть систем баз знаний и логического вывода, или функционально полиме МБД коммерческого типа для экономических приложений в задачах управления, где осуществляется доступ к структурированным данным из прикладной программы, функционирующей в главной ЭВМ.

## 5.2. Ассоциативные параллельные процессоры

*Е. К. Гордиенко*

### Основные положения

Память, адресуемая по своему содержимому (ПАС), или параллельные процессоры (ПП), адресуемые по своему содержимому, или, как принято говорить, *ассоциативная память* (АП), или *ассоциативные параллельные процессоры* (АПП) изучаются с середины 50-х годов. Ассоциативную систему типа STARAN иногда относят и к «системам с ансамблем процессоров».

«Адресуемый по своему содержимому» означает, что отдельная ячейка памяти содержит данные, которые удовлетворяют некоторому критерию.

В качестве примера АП опишем действия профессора, пытающегося выяснить, кто из его студентов имеет книгу под названием «Архитектура вычислительных машин» [Фостер, 1981]. Рассматривая студентов как матричную память и используя ее обычную адресную модель, он спрашивает их по очереди: «Имеет ли студент, сидящий на месте 1, эту книгу?», «Имеет ли студент, сидящий на месте 2, ...?» и т. д. Если профессор действует согласно модели типа магазинного стека, то он встает у двери и каждому студенту, проходящему мимо него, задает вопрос.

Если процессор действует согласно модели типа ассоциативной памяти, то он встает перед аудиторией и говорит: «Если вы имеете эту книгу, пожалуйста, поднимите руку». Конечно, это требует некоторых размышлений со стороны студентов, что отсутствует в обычной адресной модели и модели типа стека. Поэтому АП иногда называют интеллектуальной памятью. При этом обычно

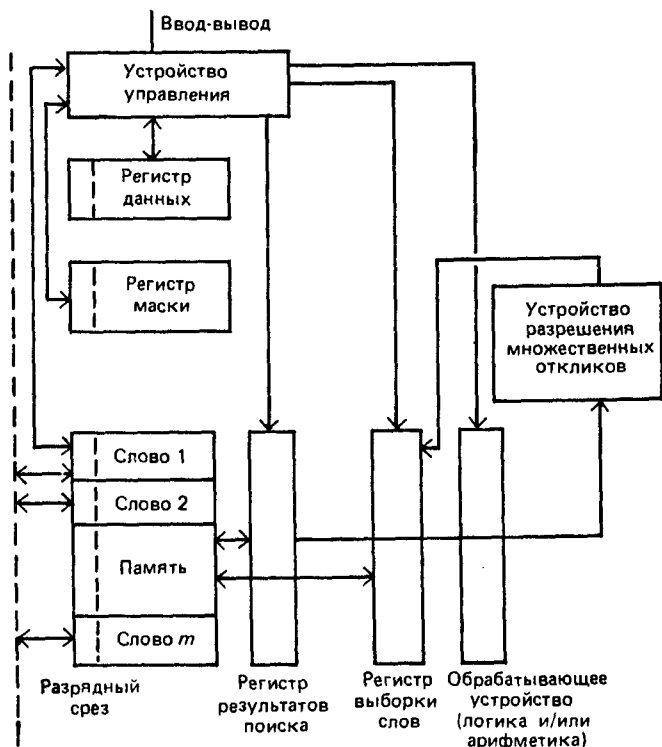


Рис. 5.14. Структурная схема АПП

предполагают, что в каждой ячейке памяти должно быть достаточное количество логических средств, чтобы дать ей возможность:

сравнить число, которое она хранит, с числом, которое посылается из устройства управления;

указать совпадение этих чисел или их несовпадение с помощью, например, состояния триггера.

Если такая АП дополняется возможностью мультizaписи в ячейки с совпадением, то такое устройство в принципе может считаться АПП.

Структурная схема типичного АПП [Тербер, 1985] показана на рис. 5.14. Этот ОКМД-процессор построен на базе ассоциативного запоминающего устройства, состоящего из ассоциативных ячеек памяти. Данные могут быть поставлены по некоторому критерию (равно, меньше чем и т. д.) с информацией, хранящейся в памяти, для чего выполняются следующие действия: 1) запись в *регистр данных*; 2) выделение разрядов, подлежащих сравнению, с использованием *регистра маски*; 3) запись битового набора, описывающего искомое подмножество данных в файле, в *регистр выборки слов*. Результатом сравнения будет битовый набор в *регистре результатов поиска* с указателем на «первое откликнувшееся слово». Этот указатель называется устройством разрешения множественных откликов, он указывает на «самое верхнее в памяти слово (ячейку)», удовлетворяющее критерию поиска, т. е. на «самый верхний бит» регистра результатов поиска.

Заметим, что операция записи битового набора файла в регистр выборки слов не обязательна, если каждый разряд любой ячейки памяти дополнен достаточным количеством логических средств для сравнения своего содержимого с содержимым соответствующего разряда регистра данных и передачи результата сравнения в регистр результатов поиска. В этом случае АП представляет собой вариант *памяти с распределенной логикой*.

Ячейка АП является основным элементом, из которых строится система. При желании модель ассоциативного процессора может быть создана и без специальных устройств, но при этом снизится быстродействие. Процессорное оборудование (процессорный элемент), обеспечивающее соответствующую аппаратную поддержку, включает по крайней мере одно полное последовательное арифметическое устройство на каждое *слово ассоциативной памяти*.

Особенностью АПП является то, что применяемое в нем запоминающее устройство (ЗУ) использует аппаратно (микропрограммно) реализованный *ассоциативный* (возможно, наряду с другими) метод доступа. Некоторые авторы [Тербер, 1985] считают особенностью АПП отсутствие взаимных связей между процессорными элементами, реализующими операции сравнения (с информацией регистра данных) в соседних ячейках памяти. Однако в настоящее время имеется устойчивая тенденция к снятию этого ограничения и сближению АПП с другими типами ОКМД-архитектур.

Не все задачи пользователей можно решить с помощью универсальных вычислительных машин (по крайней мере, за приемлемое время). Имеются задачи, требующие очень высокой скорости вычислений, обеспечение которой с помощью универсальной вычислительной системы нецелесообразно с точки зрения стоимости и высокой пропускной способности при обработке структурированных данных (например, задачи противовоздушной обороны). При решении таких задач с успехом могут быть использованы ассоциативные вычислительные системы.

Ассоциативные параллельные процессоры являются специализированными вычислительными машинами. На них могут решаться различные задачи, но метод решения должен быть один и тот же. Благодаря наличию повторяющихся структурных элементов они обеспечивают возможность эффективного (с точки зрения стоимости) применения СБИС-технологии.

Такие процессоры могут быть полезными при обработке метеорологических данных, обработке в реальном масштабе времени радиолокационных сигналов, в противоракетной и противовоздушной обороне, при управлении воздушным движением, в механизмах виртуальной памяти, механизмах защиты в операционных системах, при решении систем дифференциальных уравнений [Кохоен, 1982; Тербер, 1985; Фостер, 1981]. Наряду с вышеупомянутыми возможной областью применения (с высокой эффективностью) является использование АПП в качестве спецпроцессоров для машин баз данных (МБД) и машин баз знаний (МБЗ). В дальнейшем это последнее применение АПП будет рассмотрено подробнее.

К числу основных недостатков АПП можно отнести следующие [Berga et al., 1979]: 1) значительные дополнительные затраты на аппаратную поддержку *поисковых операций АПП*; 2) сравнительно низкие скорости обмена информацией между внешними устройствами (например, дисками) и АПП. Специализация АПП также считается недостатком.

Однако некоторые из отмеченных недостатков постепенно будут устраняться с развитием СБИС-технологии и разработкой новых перспективных устройств памяти, например памяти на цилиндрических магнитных доменах, а также памяти на базе приборов с зарядовой связью.

### Краткая история

Ассоциация может рассматриваться как установление некоторого соответствия между объектами. Аристотель определил три вида ассоциаций: по сходству, по контрасту, по близости. Затем Лок и Хью дополнили этот список ас-

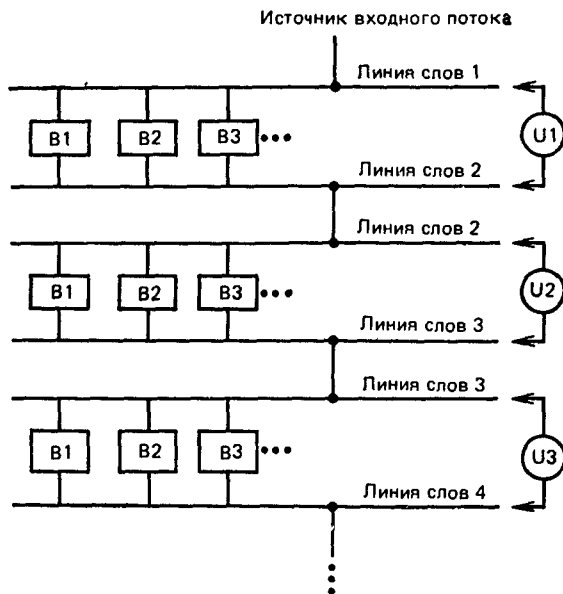


Рис. 5.15. Аппаратная часть АПП

социацией идей, а также ассоциациями по причине и следствию. Возникшее позже философское учение об ассоциациях утверждало, что знание извлекается из отдельного простого практического опыта, и поэтому оно может быть представлено в терминах таких основных опытов. Более поздние исследования механизма ассоциаций проводились на основе биологических наук, изучения нервной системы и деятельности мозга. Обширный обзор истории развития учения об ассоциациях и связанных с ними философских концепций содержится в работе [Замеpek, 1971].

Впервые концепция ассоциативного процессора была представлена в виде гипотетической системы MEMEX [Buch, 1945; Тербер, 1985]. Описаны лишь ее фундаментальные характеристики, каких-либо попыток рассмотреть практическую реализацию систем не предпринималось. В первом ассоциативном процессоре предполагалось хранить данные в виде микрофильмов и воспроизводить их на особых графических терминалах с помощью сухого фотографирования.

Первое аппаратное устройство для ассоциативной обработки было продемонстрировано в виде криогенной каталоговой памяти. Ее появление представляло собой важный этап в истории развития ассоциативной техники. Это устройство создавалось для работы в системе, которая представляла следующие три требования: хранение большого количества слов фиксированной длины; ассоциативный метод доступа к памяти; небольшое число операций стирания или записи информации по сравнению с числом требуемых операций по обобщению (собственно обработке) данных. Схемы разрешения множественных совпадений (т. е. выборки из множества ответчиков, удовлетворяющих критерию поиска, одного ответчика) не вводились, так как в предполагаемых применениях данные в памяти действительно размещались по принципу каталога. В процессор вводилась информация, характеризующая объект, и в ответ выдавался признак, показывающий, находится ли объект в каталоге. Аппаратные средства, обеспечивающие получение результатов поиска, включали вольтметры, соединенные с кодовыми шинами слов.

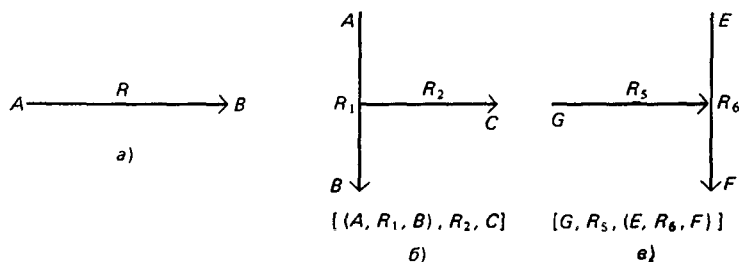


Рис. 5.16. Примеры объектов языка ПАБД

На рис. 5.15 представлена схема ассоциативного процессора (см. также [Тербер, 1985]). Входной поток вводился в систему по шинам слов. Каждый разряд ( $B_1, B_2, B_3, \dots$ ) каждого слова процессора представлял собой криотронное устройство, способное хранить и откликаться на бит информации.

Входное слово сравнивалось с содержанием каждого слова памяти путем поразрядного сопоставления. В каждой разрядной позиции, где разряд входного слова совпадает с разрядом слова из памяти, у криотрона появляется сопротивление. Если входное слово целиком соответствует слову из памяти, то сопротивление появится во всех разрядах. Факт совпадения отмечался вольтметром, подключенным к информационным шинам того слова, в котором произошло совпадение. Данная операция сравнения выполнялась одновременно со всеми разрядами всех слов.

В 1972 г. фирма Goodyear Aerospace Corporation сконструировала первый современный коммерческий ассоциативный параллельный процессор STARAN. С 1956 по 1972 гг. опубликовано свыше 100 статей по ассоциативным системам (см., например, [Minker, 1971]), но большинство проектов до реализации не доведено.

С точки зрения приложений ранних разработок к решению задач искусственного интеллекта интересны основные моменты концепции Процессора с Ассоциативной Базой Данных (ПАБД) [Тербер, 1985; Кохонен, 1982]. При разработке ПАБД использовались методы погружения сети отношений (вариант семантической сети) в вычислительную среду, использующиеся с модификациями в ряде современных систем, причем не только ассоциативных, и организации связей между процессорными элементами (в данном случае это процессоры ячейки ассоциативной памяти), для организации параллельной обработки нескольких фрагментов семантической сети.

Основой ПАБД является конструкция языка, именно поэтому язык ПАБД был разработан раньше, чем аппаратная часть. В конструкции языка для ПАБД заложены два существенных момента. Во-первых, данные и процессы создаются из упорядоченных троек объектов  $(A, R, B)$ . Эти тройки задают отношения и могут быть интерпретированы следующим образом:  $A$  к  $B$  находится в отношении  $R$  (рис. 5.16, а). Во-вторых, все процессы могут быть представлены в виде структур, описывающих совпадения (отклики) и замещения. Каждая команда ПАБД (рис. 5.17, а) состоит из двух структур данных. Одна служит для управления и представляет собой спецификацию подструктур (вплоть до отдельных отношений), которые нужно найти в структуре данных, вторая — для организации замещения и представляет собой спецификацию структур данных, подлежащих записи на место множества данных, для которых операция проверки на совпадение дает положительный результат.

Две структуры считаются совпадающими в том и только в том случае, когда соответствующие объекты и связывающие метки обеих структур совпадают. Структура данных, удовлетворяющая условию проверки на совпадение, подлежит замещению. Объекты могут быть составными, как показано на

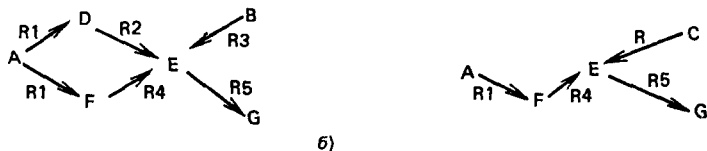
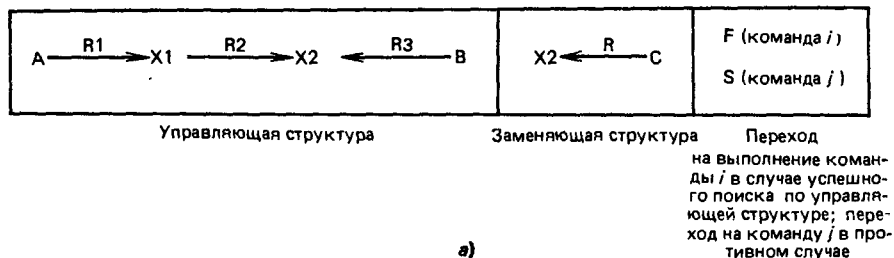


Рис. 5.17. Команда ПАБД:

$a$  — формат;  $б$  — результат применения до и после проверки на совпадение

рис. 5.16,а. Таким образом, структуры данных могут быть представлены в виде ориентированных графов (см. рис. 5.17,б). Язык ПАБД не лимитирует размер или сложность графа. В команде ПАБД может быть специфицирована операция, которую нужно выполнить для того, чтобы процессор остановился по совпадению, по несовпадению или по какой-либо другой причине (с возвратом управления в вызывающий процесс, если речь идет о подпрограмме). Разрешаются структуры, состоящие из пустого множества.

Символ  $X$  используется для обозначения неизвестных объектов, участвующих в отношении; он может быть любым объектом или связующей меткой (отношения). Неизвестные объекты могут быть специфицированы в терминах контекста отношения, в котором они найдены. Определенное выше условие совпадения подразумевает прямое совпадение, поскольку это условие устанавливает взаимно однозначное соответствие. Существуют также средства задания непрямого совпадения, но здесь они не рассматриваются.

Обобщенная схема ПАБД изображена на рис. 5.18. Главным элементом этих машин является *контекстно-адресуемая память*. Регистр маски служит для маскирования разрядных позиций данных, находящихся в регистре сравнения. В систему входят центральное устройство управления (УУ), выступающее в качестве ведущего процессора, и контроллеры с микропрограммным управлением, связанные с контекстно-адресуемой памятью. Арифметическое устройство (АУ) обеспечивает выполнение ряда специальных аппаратно реализованных функций, используемых для облегчения процесса выполнения программ.

Существует два варианта организации контекстно-адресуемой памяти ПАБД, соответствующие двум вариантам хранения сети отношений (см. рис. 5.16):

предложениями, т. е. совокупностями отношений (рис. 5.19);

объектами (идентификаторами и совокупностями указателей адресов).

В первом случае каждое слово памяти разбито на поле отношения и поле объекта. В поле отношения содержится метка связи (см. рис. 5.19,а). Начало предложения отмечено специальным сочетанием PS, находящимся в поле метки. Объект, ассоциированный с предложением, помещается в поле объекта того же слова, где находится PS. После этого детально описываются все отношения, в которых участвует рассматриваемый объект. С помощью операций ПАБД одновременно просматриваются все ячейки ассоциативной памяти, од-

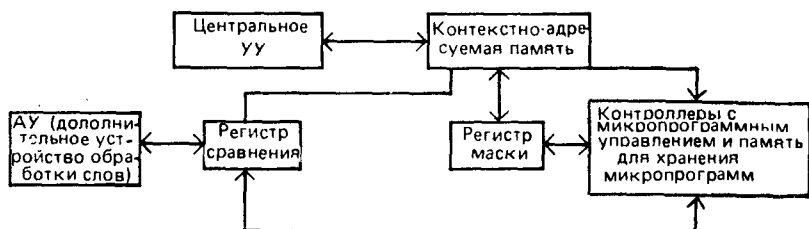


Рис. 5.18. Схема ПАБД

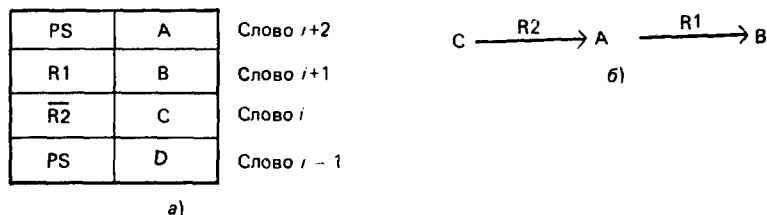


Рис. 5.19. Организация хранения данных в ПАБД, ориентированном на работу с предложениями:

а — структура памяти; б — пример предложения, хранящегося в словах  $(i-1)-(i+2)$

нако при этом обрабатывается только часть одного из описываемых управляющей структурой отношений, например часть  $(R1, B)$  отношения  $(A, R1, B)$ , находящаяся в слове  $(i+1)$ . Передавая результаты обработки данной части отношения в соседнюю ячейку памяти и повторяя обработку хранящейся в этой ячейке следующей части отношения, можно обработать все отношения, входящие в предложение.

Структура второго варианта памяти ПАБД позволяет выполнять полную обработку ее содержимого параллельно, т. е. одновременно для любого числа заданных отношений.

На рис. 5.20 изображен небольшой массив контекстно-адресуемой памяти ПАБД, содержащий информацию (рис. 5.17). Заданы адреса ячеек памяти, состоящие из двух координат  $(r, c)$ , где  $r$  — номер строки, а  $c$  — номер столбца. Каждая ячейка содержит три поля данных одинакового размера и поля для флажков. Если в ячейке хранятся литеральные константы или идентификаторы связей, все три поля можно объединить. Если ячейка должна содержать тройку данных, т. е. отношение или составную конструкцию, ее элементы задаются косвенно с помощью адресных кодов, указывающих на положение в памяти соответствующих литеральных констант или идентификаторов связи. При таком способе представления данных содержимое трех описанных выше полей ячейки эквивалентно указателям. Если тройка содержит неизвестные элементы, для них необходимо использовать специальные резервные коды. Содержимое ячеек массива ПАБД для рассмотренного примера (рис. 5.17) показано на рис. 5.20. Для пояснения под ячейками помещено символическое обозначение содержимого.

Каждая ячейка кроме содержимого трех полей может также передавать свой собственный адрес. Любой из четырех адресов ячейки можно включить и передать в линию связи при наличии соответствующего сигнала глобального управления. Следует отметить, что такое включение и передача возможны только для тех ячеек, которые имеют в момент передачи определенные сочетания состояний управляющих триггеров.



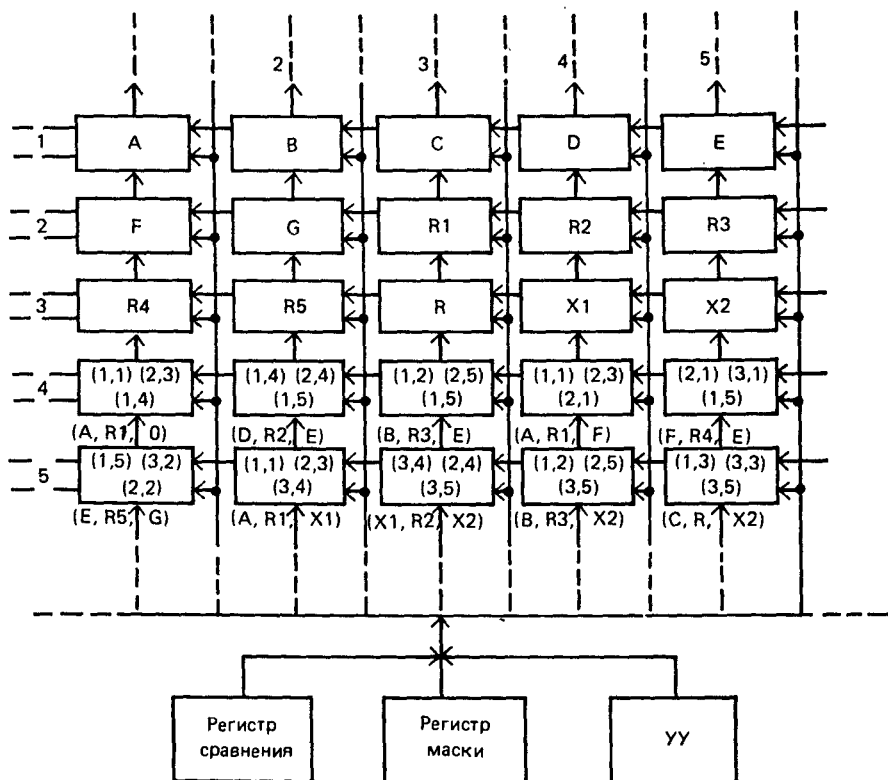


Рис. 5.20. Заполненный массив контекстно-адресуемой памяти ПАБД, ориентированного на работу с объектами

Ячейки также могут пропускать через себя адресные коды снизу вверх (с юга на север) и справа налево (с востока на запад). Коды, передаваемые на запад, являются копией кодов, вошедших в ячейку с востока или с юга. Если коды идут с востока, их передвижение на запад выполняется автоматически до тех пор, пока принимаемый какой-либо ячейкой код не совпадает с адресом ячейки. В этом случае дальнейшее продвижение кода прекращается и включается триггер совпадения этой ячейки. Если код передается с юга, то при совпадении его адреса строки с адресом строки очередной ячейки он автоматически будет направлен на запад. В противном случае код будет передаваться дальше на север.

Ячейки, находящиеся на краях массива, связаны с ячейками, находящимися на противоположном краю массива, поэтому массив имеет форму тора. Таким образом, сигналы из крайних верхних ячеек, которые необходимо переместить на север, попадают в крайние нижние (южные) ячейки. Аналогичные связи существуют и между крайними левыми и крайними правыми ячейками.

Поскольку в массиве одновременно перемещается большое число кодов, существует вероятность того, что коды, передаваемые с южного и восточного направлений, одновременно достигнут одной ячейки. В этом случае может возникнуть конфликтная ситуация, если перемещаемый на север код необходимо отправить на запад, поскольку ячейка одновременно может пропустить только

один код в этом направлении. Такая ситуация называется блокировкой, и в этом случае приоритет будет иметь код, пришедший с востока. Код, пришедший с южного направления, сначала пропускается в ближайшую северную ячейку, после чего возвращается в ячейку, где возникла конфликтная ситуация. Если массив не слишком заполнен, то такой дополнительный цикл не приведет к существенному снижению общей скорости обработки данных.

Система из восьми микроопераций над памятью ПАБД включает операции считывания-записи, ассоциативные операции поиска (АОП), например сравнение информации, находящейся в ячейках, с информацией в регистре сравнения. Кроме того, имеются операции, реализующие связывание информации об отдельных идентификаторах отношения, а также связывание информации, полученной при обработке нескольких отношений управляющей структуры.

Эта система микроопераций позволяет работать со сложными управляющими структурами, причем отношения обрабатываются в определенном поряд-

ке. Если, например, имеется управляющая структура  $A \xrightarrow{R_1} X \xrightarrow{R_2} E$ , то сначала обрабатываются два отношения  $(A, R_1, X)$  и  $(X, R_2, E)$ , затем находятся ячейки, которые содержат значение  $X$  (т. е.  $D$  — см. рис. 5.17,б), удовлетворяющее конъюнкции первых двух обработок. Каждая команда ПАБД реализуется в виде совокупности микроопераций, в процессе выполнения которых возможны значительные перемещения информации между ячейками контекстно-адресуемой памяти.

Чтобы ускорить перемещение кодов в массиве, ПАБД предполагалось дополнить следующим средством. Небольшое число ячеек ( $\sqrt{N}$ , где  $N$  — общее число ячеек), распределенных в памяти большой емкости, предполагалось использовать для прямой передачи сигналов от одной ячейки к другой, т. е. для «перескока» через ячейки, лежащие между ними.

Очевидно, что для реализации всех микроопераций структура ячеек массива памяти ПАБД должна быть существенно более сложной, чем в классической схеме АПП. Число управляющих шин и линий также весьма велико. Однако существенно то, что это позволяет достичь большой степени параллелизма при выполнении как глобальных, так и локальных операций. Более того, поскольку внутренняя структура ячеек является регулярной и поэтому перспективной для реализации в виде СВИС, принципы такой организации памяти можно считать альтернативными для принципов, заложенных в ЭВМ фон-неймановского типа. Некоторые принципы организации памяти ПАБД уже нашли практическое воплощение в современных сверхвысокопроизводительных вычислительных системах, например в Colpaction Machine [Hillis, 1985].

### Классификация АПП

С точки зрения обработки информации ассоциативная система может выполнять сложные операции преобразования данных в дополнение к операциям сравнения, которые может выполнять ее ассоциативная память. С точки зрения структуры ассоциативные системы входят в класс ОКМД [Головкин, 1980] и в свою очередь, могут быть разбиты на четыре категории в соответствии с организацией процесса сравнения в их ассоциативной памяти: 1) полностью параллельные, т. е. параллельные и по разрядам (или группам разрядов), и по словам; 2) параллельные по словам и последовательные по разрядам; 3) по слову последовательные; 4) блочно-ориентированные ассоциативные системы. Опишем основные принципы функционирования систем каждого типа. Более детальное изложение особенностей конкретных систем содержится в [Головкин, 1980; Тербер, 1985].

В полностью параллельных системах логика сравнения может быть предусмотрена в каждой ячейке разряда каждого слова или же в каждой группе ячеек (например, в байте), представляющих код символа при фиксированном числе разрядов в коде (или в группе кодов, например, в слове). С точки зрения ассоциативного характера обработки к полностью параллельным ассоциа-

тивными системам с логикой сравнения в группе разрядов можно отнести систему РЕРЕ [Головкин, 1980; Тербер, 1985]. Наибольшую производительность могли бы иметь полностью параллельные системы с логикой сравнения в каждом разряде, но оборудование в таких системах сложное и дорогостоящее, поэтому они не нашли широкого применения.

Пословно-последовательная ассоциативная система фактически представляет собой аппаратную реализацию простого программного цикла для поиска. Главный фактор, способствующий относительно более высокой эффективности этого подхода по сравнению с запрограммированным поиском в обычной ЭВМ, заключается в том, что уменьшается время декодирования команды, поскольку в пословно-последовательном процессоре требуется только одна команда для выполнения операции поиска. Такие процессоры могут быть реализованы на основе вращающегося устройства ассоциативной памяти, цифровых ультразвуковых линий задержки, барабанов или дисков с логикой сравнения на каждый тракт. Вследствие низкой скорости пословно-последовательной ассоциативной памяти были предложены только экспериментальные модели пословно-последовательных ассоциативных систем.

Блочно-ориентированную ассоциативную систему можно рассматривать как компромисс между сравнительно дорогостоящими поразрядно-последовательной и параллельной системами и низкоскоростной пословно-последовательной ассоциативной системой. Система использует вращающееся устройство памяти большой емкости, такое как диск с ограниченными ассоциативными возможностями, имеющий головку на тракт (дорожку) и некоторую логику для каждого тракта. Примером блочно-ориентированной системы является RAPID (Rotating Associative Processor for Information Dissemination) [Parhami, 1972].

Наиболее широкое применение наряду с полностью параллельными получили поразрядно-последовательные системы (при этом эти системы являются пословно-параллельными). В таких системах обычно используется схема, показанная на рис. 5.21. Поразрядно-последовательный подход может быть реализован путем использования запоминающего устройства с произвольным доступом и неразрушающим чтением, в котором направление расположения информации повернуто на 90° по сравнению с обычными устройствами памяти. Слово обычной памяти образует битовый срез ассоциативного процессора. Размеры запоминающего устройства желательно выбирать такими, чтобы ассоциативный процессор имел значительно больше слов ассоциативной памяти, чем битовых срезов. Тогда стоимость реализации такого подхода, без учета стоимости внешнего оборудования, может стать близкой к стоимости памяти с произвольным доступом.

Все операции выполняются с битовыми срезами с помощью внешнего устройства обработки слов. Если каждую ячейку ассоциативной памяти можно было бы снабдить аппаратными средствами для выработки сигналов наличия/отсутствия отклика, то это увеличило бы быстродействие системы с ассоциативной обработкой. Обычно сначала происходит считывание битового среза и затем во внешних по отношению к памяти цепях генерируются сигналы наличия/отсутствия отклика. Таким образом, слова обычного оперативного запоминающего устройства становятся битовыми срезами АПП.

Если по каким-либо причинам невозможно использовать память с неразрушающим считыванием, то во избежание потерь информации вводится цикл «чтение/восстановление». В этом случае ввод или вывод слов обычно производится в поразрядно-последовательной форме. Внешние логические элементы и устройство запоминания откликов могут состоять из одного сигнального разряда на каждое ассоциативное слово и схемы поэтапного выбора слов (т. е. разрядных срезов), а также последовательных сумматоров (по одному на каждое ассоциативное слово). В состав внешнего оборудования также могут быть включены каналы и средства ввода-вывода. Одним из возможных вариантов организации ввода-вывода является параллельный канал, с помощью которого выводятся или вводятся битовые срезы.

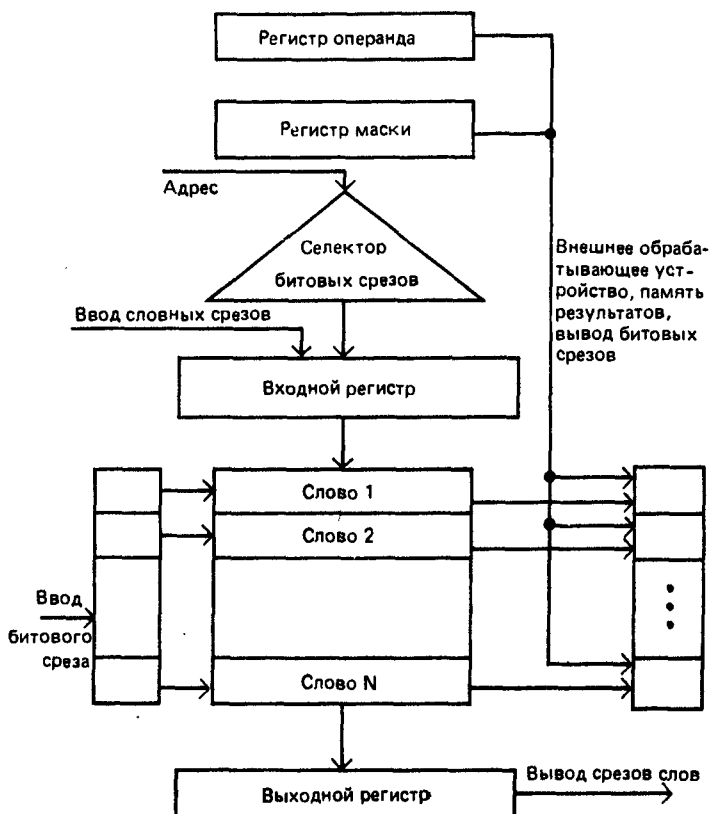


Рис. 5.21. Поразрядно-последовательный АПП

Основной поиск на равенство осуществляется поразрядно-последовательно. Поиск на равенство с полем, содержащим  $n$  разрядов, выполняется за  $n$  циклов обработки битовых срезов. Принцип обработки битовых срезов был использован в системе STARAN [Головкин, 1980; Тербер, 1985].

Необходимо отметить, что все АПП с параллельной по словам обработкой или блочно-ориентированные принадлежат к классу систем с вертикальной обработкой, основной принцип которой состоит в применении операций, в которых в качестве аргументов используются не строки массива информации (слова), а его столбцы или группы столбцов [Фет, 1986].

#### Построение спецпроцессоров для машин баз данных с применением АПП

Выполнение ассоциативных поисковых операций («точное совпадение с образцом поиска», «больше, чем», «меньше, чем» и др.) является основой для реализации любого запроса в Базе Данных (БД). Поэтому аппаратная реализация этих и других операций над данными в БД (например, операции соединения) представляется весьма перспективной.

Обычно машины БД (МБД) определяются как комплексы микропрограммных и аппаратных средств для выполнения некоторых или всех функций уп-

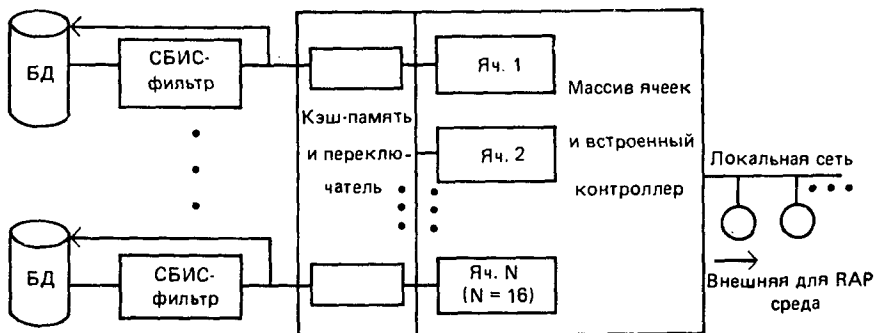


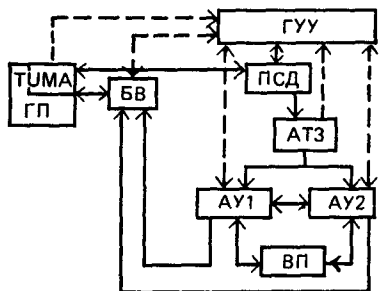
Рис. 5.22. МБД RAP.3

правления базой данных [Гордиенко и др., 1985]. На рис. 5.22 показано возможное место МБД в вычислительной системе (ВС). В ВС без МБД алгоритмы выполнения запросов в БД реализуются как программы главных процессоров (ГП), а в МБД для выполнения (полного или частичного) запроса используется специальная промежуточная обработка данных, аппаратно реализуемая на периферийном оборудовании, которое по существу и представляет собой машину базы данных. Архитектура МБД определяется строением собственно базы данных и ее системы управления на основе использования моделей представления данных и языков описания и манипулирования данными.

Ассоциативные параллельные процессоры используются в качестве спецпроцессоров для МБД благодаря двум важным достоинствам: ассоциативной адресации и параллельной обработке.

С точки зрения рассмотренной выше классификации известные АПП для МБД относятся либо к блочно-ориентированным (системы CASSM, DBC, RAP [Гордиенко и др., 1985; Berra, Oliver, 1979]), либо к поразрядно-последовательным (система RELACS [Oliver et al., 1979]). Представляется целесообразным рассмотреть две системы: RELACS, поскольку в этой системе используется наиболее традиционный тип архитектуры АПП — поразрядно-последовательный, и RAP, поскольку последние ее версии в значительной мере отражают тенденции развития АПП для МБД (например, возможность использования ПЗС и памяти на ЦМД в качестве основы для построения ассоциативной памяти).

**Система RELACS** (Relational Associative Computer System). Эта система представляет собой периферийную машину баз данных, создающую для поддержки реляционной модели БД. Как показано на рис. 5.23, система состоит из следующих основных функциональных блоков: главное управляющее устройство (ГУУ); процессор словаря данных (ПСД); ассоциативный транслятор запроса (АТЗ); ассоциативные устройства АУ1 и АУ2; вторичная (массовая) память (ВП) и буфер вывода (БВ).



Запросы пользователя проходят обработку в главном процессоре (ГП) перед поступлением в систему RELACS (на рис. 5.23 потоки данных указаны сплошными линиями, а управляющие связи — штриховыми). Проводится грамматический разбор запроса, выполняются синтаксические функции и определяется порядок

Рис. 5.23. Архитектура системы RELACS

поиска атрибутов и/или отношений; при этом логические связи запроса сохраняются. Сформированные затем последовательность команд запроса и управления хранится в специальной области памяти ГП, обозначаемой TUMA (Transfer Users Memory Area). При этом обслуживание заданий организуется по принципу FIFO и используется более сложная дисциплина обслуживания с применением приоритетов. После того, как задание помещено в TUMA, ГП передает об этом сообщение в ГУУ.

Обработка запроса начинается в ПСД, представляющем собой группу соединенных массивов ячеек ассоциативной памяти. Здесь прежде всего выясняется, имеются ли имена атрибутов и/или отношений в БД. Если имеются, то проверяется соответствие запроса праву пользователя на доступ к данной информации и уровень его привилегий к указанным атрибутам и/или отношениям.

Ассоциативный транслятор запроса выполняет функции трансляции исходной последовательности инструкций пользователя и дескрипторных данных (передаваемых ПСД) в команды для их исполнения ассоциативными устройствами АУ1 и АУ2. При этом подключается массив ассоциативной памяти, содержащий запрос и дескрипторные данные, необходимые для трансляции, а также ЗУ с произвольным доступом, хранящее дескрипторные данные всех отношений, и стек памяти или буфер, который загружается командами для ассоциативных устройств по мере того, как формируются эти команды.

Ассоциативные устройства АУ1 и АУ2 обращаются к АТЗ для формирования команд обращения к ВП с целью получения требуемого отношения, по которому нужно провести поиск и/или обновление, специфицированное запросом, и выполнить необходимую задачу. Главными компонентами АУ1 и АУ2 являются массивы ассоциативной памяти. В их состав также входят: массив компарандов для множества аргументов поиска; массив отклика (т. е. аппаратных средств, фиксирующих результаты поиска); буфер вывода и средства ввода-вывода. При этом отношения реляционной модели данных отображаются в структуру массивов ассоциативной памяти (например, кортеж отношения может отображаться в слово массива). Вывод из ассоциативного устройства производится либо в ВП (обновление), либо пользователю через буфер вывода и ГП, либо другому ассоциативному устройству (при выполнении операции JOIN).

**Система RAP** (Relational Associative Processor). Эта система прошла значительный путь развития, начиная с SIMD-архитектуры (использующей диски с фиксированными головками и сравнительно простые процессорные элементы), ориентированной на реляционную модель данных. В настоящее время, сохраняя основные концепции и используя достижения перспективных подходов к построению устройств памяти, RAP представляет собой MIMD-систему [Ozkanagan, 1985], способную поддерживать БД с различными моделями данных.

*Основные концепции системы RAP.*

1. Архитектура всех версий этой системы представляет собой МБД с ячейочной (cellular) ассоциативной организацией (см. рис. 5.22, 5.24), причем в качестве основы для построения ячеек используется память массового производства.

2. Структурами данных в RAP являются специальные нормализованные отношения практически любой арности. В дополнение к атрибутам данных RAP-отношение содержит также теговые атрибуты (маркирующие биты) для эффективной реализации различных ассоциативных операций без переупорядочивания временных или рабочих отношений.

3. Реляционные операции языка манипулирования данными RAP-системы образуют в совокупности реляционно полный язык и могут быть двух типов: унарные, преобразующие заданное подмножество (кортежей) отношения в другое множество кортежей, и бинарные, реализующие отображения пары реляционных подмножеств в некоторое новое множество. Программа для реализации запроса представляет собой последовательность операций.

Далее кратко рассмотрим особенности одной из последних версий системы — RAP.3 (см. рис. 5.22).

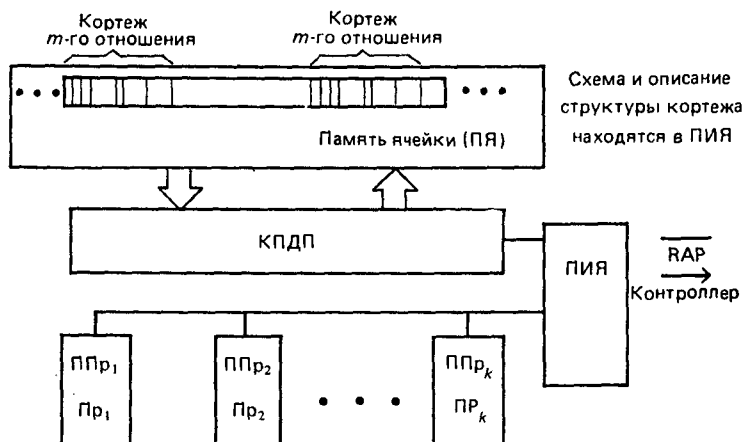


Рис. 5.24. Структура ячейки системы RAP.3

На рис. 5.24 показана структура ячейки (Яч), представляющей собой мультимикропроцессорную систему. При этом  $K$  микропроцессоров  $Пр_i$  (в существующей реализации  $K=4$ ), именуемых субъядчейками, работают параллельно; кроме того, имеется процессор интерфейса ячейки (ПИЯ), осуществляющий общую координацию компонентов ячейки. В процессе работы кортежи (или даже только нужные для RAP-операции атрибуты этих кортежей) обрабатываются отношениями считываются (загружаются) из памяти ячейки (ПЯ) пословно с помощью контроллера прямого доступа к памяти (КПДП). Обмен информацией между  $Пр_i$  и ПЯ осуществляется таким образом, что на время передачи данных  $Пр_i$  блокируется в цикле ожидания и разблокируется контроллером КПДП, когда кортеж помещается в память процессора ( $ППр_i$ ). Считывание и запись кортежей в ПЯ осуществляется в расщепленном цикле доступа к памяти.

Во время исполнения операции субъядчейка повторяет выполнение программы процессора  $Пр_i$ , реализующей эту операцию, над каждым кортежом, полученным этой субъядчейкой в свою  $ППр_i$ . При этом из  $k$  процессоров  $Пр_i$  ( $k=2$ ) осуществляют обработку считанных кортежей, а два оставшихся — ввод-вывод кортежей, причем множества обрабатывающих и осуществляющих ввод-вывод  $Пр_i$  меняются циклически.

Емкость ПЯ составляет до 2 Мбайт динамической памяти с произвольным доступом. Каждый  $Пр_i$  представляет собой микропроцессор Intel 8086, а  $ППр_i$  может содержать программы обработки RAP-операций, обрабатываемый кортеж, а также значения (до 1500 символов), которые используются для сравнения с элементами кортежа в процессе выполнения RAP-операции. Все ячейки функционируют под управлением встроенного контроллера, осуществляющего также взаимодействие с внешней для RAP средой пользователей в форме (запрос, ответ). В имеющихся версиях контроллер строился на базе iSBC86/12.

Как видно из рис. 5.22, на пути между массивом ячеек архивной памяти (диски) имеется кэш-память и переключатель, что позволяет организовать различные стратегии переноса данных между ЗУ различных уровней. Кроме того, между дисками и кэш-памятью имеются ассоциативные СБИС-фильтры данных. Каждый такой фильтр реализован на базе одной (или двух, если используется и микросхема FIFO-буферной памяти) микросхемы — простого процессора, способного осуществлять оперативным способом исполнение унарных реляционных

операций. Использование СБИС-фильтров в реализации ряда RAR-операций дает значительный дополнительный временной выигрыш.

Необходимо отметить, что значительный выигрыш аппаратная реализация выполнения запросов к БД по сравнению с программной будет иметь лишь в том случае, если большинство выполняемых запросов будут использовать сложные операции (такие, как JOIN-операция), требующие обработки нескольких отношений (массивов), или операции, требующие выполнения агрегативных функций [Гордиенко и др., 1985].

### **Основные тенденции развития АПП**

Развитие аппаратных средств, использующих ассоциативный принцип обработки информации, определяется, с одной стороны, успехами новых технологичных построения устройств памяти, а также совершенствованием ОКМД-архитектур, с другой — достижениями в области построения алгоритмов для ОКМД-систем в традиционных и новых сферах их применения. По-видимому, наиболее перспективны следующие направления развития АПП.

1. Построение АЗУ на новой элементной базе: приборы с зарядовой связью (ПЗС); ЗУ на цилиндрических магнитных доменах (ЦМД); приборы, использующие оптические методы обработки информации, и др. [Кохонен, 1982].

2. Использование массивов доступных полупроводниковых БИС (СБИС) в качестве основы для построения АПП и перенесение активности в исследованиях на алгоритмы и микропрограммную (либо макрокомандную) реализацию процедур поиска с применением ассоциативных поисковых операций [Hillis, 1985; Ozkarahan, 1985].

3. Использование аппаратных фильтров предварительной ассоциативной обработки информации на пути из архивной памяти (диски) в основную вычислительную систему [Ozkarahan, 1985].

4. Введение аппаратно реализованных локальных связей между ассоциативными процессорными элементами и интенсивное использование этих связей в алгоритмах обработки информации [Гордиенко, 1986; Hillis, 1985]. При этом в качестве процессорных элементов могут использоваться как мелкоячеистые спецпроцессоры с емкостью внутренней памяти в несколько килобит (4 Кбит в системе [Hillis, 1985]), так и крупноячеистые — с памятью в несколько килобайт (системы на базе транспьютеров) и более.

5. Применение ассоциативных МБД для работы непосредственно со структурами знаний (а не данных), представляемых с помощью семантических сетей или сетей фреймов [Гордиенко, 1986; Hillis, 1985].

### **5.3. Однородные структуры**

*В. Н. Захаров, В. Ю. Кириллов*

#### **Основные положения**

Проблема поиска высокопроизводительных структур вычислительных систем, которые могли бы составить основу аппаратной поддержки процессов обработки информации в интеллектуальных системах, неизбежно приводит к анализу возможностей многопроцессорной распределенной обработки информации. Поскольку быстродействие существующих вычислительных машин с последовательным выполнением программ приближается к физическому пределу скорости распространения электрических сигналов, дальнейший рост производительности вычислительных машин связан с необходимостью изменения их архитектуры в основном в направлении организации многопроцессорных вычислительных систем с широким использованием параллелизма при обработке информации. Успехи микропроцессорной, технологии создают все предпосылки



для реализации распределенных многопроцессорных систем. Одним из перспективных классов многопроцессорных архитектур являются однородные структуры.

Под однородной структурой подразумевается вычислительная сеть, состоящая из одинаковых вычислительных элементов, связанных друг с другом регулярным образом. В частности, однородной структурой является конечная сеть, образованная повторением некоторого элемента (или связанной группы элементов) со всеми его связями так, что выходы этого элемента становятся входами аналогичного элемента (или элементов). Повторяющийся элемент называется модулем вычислительной сети с однородной структурой, а сама сеть — однородной средой (ОС). Отличительной особенностью ОС является способность объединения вычислительных ресурсов всех модулей для решения одной задачи [Евреннов и др., 1966].

Разработка ОС связана с определением такой внутренней структуры каждого модуля и связей между ними, при которой ОС в целом могла бы рассматриваться как высокопроизводительная вычислительная система, удовлетворяющая определенным требованиям быстродействия, надежности, стоимости и т. д. В общих чертах требования к базовому модулю и его связям в ОС, ориентированной на применение в качестве аппаратной поддержки интеллектуальной системы, остаются теми же, что и требования к базовому модулю мультипроцессорной системы или универсальной макетирующей среды (см. § 4.2). Применительно к однородным структурам их можно уточнить.

1. Требование универсальности. В однородной среде система команд базового модуля может не быть универсальной. Его функции ограничены классом решаемых в ОС задач. Поэтому в качестве такого базового модуля можно с успехом применить RISC-процессор с настраиваемой на класс решаемых задач системой команд (см. § 4.2).

2. Требование развитого интерфейса связи. В ОС соседство модулей предопределено ее конструкцией, причем число соседей, как правило, фиксировано. Необходимость близкого действия обмена информацией среди модулей, решающих одну задачу или обрабатывающих некоторую порцию данных, должна быть обеспечена настройкой связей. При этом некоторые модули могут оказаться транзитными и непосредственно не участвовать в обработке данных. Это приводит, в свою очередь, к необходимости организации каналов быстрой передачи информации, и усложняет проблему синхронизации (самосинхронизации) процессов обработки данных.

3. Требование необходимости связей с шинами. Это требование связано с необходимостью загрузки модулей среды исходной информации для решения поставленной задачи.

4. Ориентация базового модуля на специальный язык программирования. Применительно к ОС эта проблема еще не решена. Одним из перспективных направлений исследований для решения этой задачи следует считать разработку так называемых волновых языков, ориентированных на волновой способ передачи и обработки информации [Сапаты, 1986]. При таком способе сообщения между модулями ОС организуются в виде волн, распространяющихся по сети. Нетрудно видеть, что параллелизм обработки информации в такой системе может быть очень высоким.

При организации ОС, ориентированной на реализацию процессов в интеллектуальных системах, возникает немало трудностей. Одной из них является автоматическое формирование из программ пользователей заданий и их распределение по отдельным модулям [Agerwala et al., 1982]. Сложность решения этой задачи усугубляется необходимостью уравнивания нагрузок каждого модуля-процессора в ОС и каналов передачи сообщений, без чего нельзя получить высокую производительность всей системы в целом. Важно, чтобы равномерное распределение нагрузки на процессоры и каналы связи сохранялись в течение всего времени решения задачи, что очень трудно осуществить практически. Перегрузка отдельных линий связи во время решения задачи приводит к снижению пропускной способности системы, или, как отмечается в [Stone,

1987], к ее деградации из-за эффекта насыщения (утяжеление коммуникационных связей результатами решения задач отдельными процессорами). Весьма сложной задачей является организация обработки больших массивов данных со сложной структурой, например больших списков с несколькими уровнями вложенности компонентов.

При решении задачи реализации ОС на базе микропроцессоров помимо перечисленных трудностей появляются чисто технические, связанные с физическими ограничениями процессов, происходящих в реальных элементах. Синхронизация взаимодействия отдельных процессов, управление и аппаратная поддержка большого числа связей между процессорами — вот наиболее трудные технические задачи при реализации распределенных вычислительных систем.

### Краткая история

Проблема использования однородности в вычислительных структурах впервые была сформулирована фон Нейманом в 1955 г. в работах по исследованию клеточных автоматов [фон Нейман, 1971]. Структура клеточного автомата вполне отражала свойства ОС, а именно: регулярность топологии, функциональную однородность, локальность связей. Первоначально работы по исследованию функциональных возможностей однородных структур развивались в рамках теории автоматов. Основной задачей в этот период была задача погружения автоматного оператора и вычислительных алгоритмов в однородную структуру с целью их реализации.

В нашей стране в области исследования функциональных возможностей итеративных логических сетей работало несколько научных коллективов, в том числе группа под руководством проф. В. И. Варшавского [Варшавский и др., 1973] и проф. В. Г. Лазарева [Лазарев и др., 1984]. В рамках моделей вычислительных систем концепция однородности исследовалась в работе [Евреинов, 1981]. Предложенная модель коллектива вычислителей позволяла описать совместную работу совокупности одинаковых вычислительных модулей. Основными особенностями этой модели были:

- параллельность выполнения операций;

- программируемость (изменяемость) структуры, т. е. приведение связей в коллективе вычислителей в соответствие со структурой информационных связей решаемой задачи;

- конструктивная однородность, т. е. идентичность реализации каждого вычислителя, в частности способов коммуникации с другими вычислителями.

В рамках этой модели исследовались возможности реализации произвольных структур информационных связей и выполнения любых алгоритмов, в том числе параллельных (функциональная, структурная и алгоритмическая универсальность). Естественно, далеко не все алгоритмы реализуются в этой модели с максимальной эффективностью. Исследовались вопросы устойчивости к отказам вычислителей и всей сети в целом (вопросы живучести), возможности наращивания модулей с целью повышения производительности вычислительной среды.

Кроме этих работ концепция универсальных однородных вычислительных систем (ОВС) рассматривалась также в работе [Васильев и др., 1985], где предлагалось использовать ОВС для моделирования многопроцессорных систем. Для описания алгоритмов функционирования применялись так называемые управляющие сети — специальный подкласс обобщенных сетей Петри.

В работах [Каляев, 1984; Кодачигов, 1984] развивается концепция многопроцессорной вычислительной системы с перестраиваемой архитектурой, которую также можно рассматривать как ОВС. Реализация ОВС в работе [Каляев, 1984] базируется на использовании конструктивно-однородных вычислительных модулей (ВМ), имеющих собственную память и соединенных типовыми связями с коммуникационной структурой. Применение специальной коммутирующей схемы нарушает принцип локальности связей между модулями и дает возможность связывать между собой весьма удаленные ВС. Это упрощает программирование

ОВС, так как при наличии коммутатора отпадает необходимость решения задачи оптимального погружения исходных данных в ОВС (загрузки модулей с учетом топологии их связей).

В упомянутых выше работах описывались универсальные ОВС, эффективность применения которых анализировалась вне связи с возможностью решения с их помощью задач из области искусственного интеллекта. В конце 70-х годов в связи с развитием СБИС-технологии и широким применением микропроцессоров появилась реальная возможность построения высокопроизводительных, достаточно надежных и относительно недорогих универсальных ОВС. Закачивается разработка новых перспективных языков программирования для задач искусственного интеллекта и подытоживается опыт программной реализации алгоритмов решения интеллектуальных задач, поставивший со всей очевидностью вопрос о необходимости распараллеливания процессов их решения. Интерес к ОВС как к возможной архитектуре синхронизированных вычислителей для решения интеллектуальных задач резко повышается.

В работах К. Хьюитта [Hewitt, 1973] описывается ОВС, предлагаемая в качестве аппаратной поддержки процессов преобразования их формализации с применением акторов — модели механизмов обработки знаний. Модель обладает свойствами однородности, локальности, регулярности структуры (система образуется путем повторения одних и тех же базовых элементов), простоты набора операций ВМ, изотропности (ОВС одинакова по всем направлениям). Последние три свойства отвечают требованиям СБИС-технологии. Все последующие разработки ОВС учитывают их в значительной степени. В то же время в работе ставится под сомнение принцип программируемости вычислительной структуры. Это обусловлено априорной неопределенностью информационных потоков при работе со значениями. Вместо жесткой фиксации вычислительной структуры, обеспечивающей решение конкретной задачи, предлагается механизм перемещения акторов по среде для наиболее эффективной коммуникации между ними.

В [Fahlgan, 1983] предложена архитектура параллельной системы NETL для эффективных вычислений, связанных с наследованием свойств. Описанные эвристические стратегии распространения маркеров были в дальнейшем обобщены У. Хиллисом, предложившим их реализацию в ОВС, названной им коммутиционной машиной [Hillis, 1985; Хиллис, 1987]. Кроме того, У. Хиллис показал, что в предложенной им архитектуре эффективно распараллеливаются операции реляционной алгебры Кодда, что также делает весьма перспективным применение ОВС для работы со знаниями (см., например, [Нильсен, 1985]).

### Реализация ОС

В 80-х годах ОВС начали использовать в реальных вычислительных устройствах. Опыт их разработки показал, что ощутимый выигрыш от одновременной работы ВМ можно ожидать лишь при выполнении следующих двух условий: высокой степени согласованности функционирования ВМ при совместной работе и большом числе (больше 100) ВМ.

Эти условия внесли некоторые уточнения в современное понятие об ОВС. Важнейшие из них: локальность взаимодействий между ВМ (принцип близкого действия [Евреинов и др., 1966]); наличие у ВМ собственных блоков памяти с быстрым доступом; разделение ВМ на два функционально различных процессора (главный и коммуникационный, которые могут работать параллельно) [Hewitt, 1973]; реализация децентрализованного управления; доступность информации, хранящейся в памяти любого ВМ, для любого другого модуля ОС [Hillis, 1980] и, следовательно, необходимость быстрой связи удаленных модулей или быстрой передачи сообщений между ними. Для задач из области искусственного интеллекта последнее требование особенно актуально вследствие неопределенности информационных потоков между параллельными ветвями и динамически изменяющейся структуры при решении задачи.

С точки зрения аппаратного обеспечения ОС можно классифицировать по следующим признакам:

емкость памяти и функциональные возможности модулей;  
топология связей между модулями;  
режимы передачи информации между модулями.

Емкость памяти и функциональные возможности модулей. Этот признак является существенным для оценки функциональных возможностей ОС в целом. Различают мелкозернистые (fine-granular) и крупнозернистые (coarse-granular) структуры. Первые характеризуются небольшой памятью модулей (порядка 1 Кбайт), в подавляющем большинстве случаев использованием внешней синхронизации, большой долей операций, выполняемых централизованно, централизованным упорядочением информационных потоков между модулями ВМ. Решение задач на такой ОС связано с необходимостью разбиения данных и формирования объектов обработки малого объема.

Простота модулей позволяет создавать ОС, состоящие из десятков, а в будущем и сотен тысяч модулей [Hillis, 1985].

В ОС с крупнозернистой структурой модули близки по функциональным возможностям к мини-ЭВМ. Для таких ОС характерны редкие обращения модулей к центральным устройствам (как правило, они требуются лишь при вводе-выводе), асинхронная реализация, возможность поддержки модулями нескольких виртуальных параллельных процессоров, неупорядоченные и независимые информационные потоки в ОС. Реализация алгоритмов в таких ОС ориентирована на их естественный параллелизм на функциональном уровне.

В любой системе, состоящей из сотен и более модулей, безотказная работа всех компонентов не всегда достижима. Поэтому при разработке ОС значительное внимание уделяется вопросам отказоустойчивости всей системы в целом. В работах [Hewitt, 1973; Hillis, 1985] рассматривается функциональное разделение модуля ОС на два процессора: главный и коммуникационный, поскольку процессы обработки данных и пересылки сообщений разумно разделить и выполнять одновременно. Кроме того, включение в архитектуру модуля главного и коммуникационного процессоров повышает надежность его работы. Модуль организуется с использованием принципов самодиагностики, причем для установления работоспособности каналов связи используются протоколы передачи данных между модулями. При отказе модуля или канала связи соседние модули диагностируют отказ и перестраивают алгоритм передачи данных (см., например, [Pradhan, 1985]) или структуру связей ОС, так что работа ОС остается корректной при некотором снижении быстродействия (деградации структуры). Отказоустойчивость и степень деградации при отказах в значительной степени определяются топологией связей в ОС.

Топология связей между модулями. Основными критериями выбора оптимальной топологии ОС служат простота коммуникаций и простота реализации всей структуры на базе СБИС-технологии.

Наиболее простой топологией обладает так называемая торидальная ОС (матричная ОС, свернутая в тор путем соединения соответствующих периферийных элементов). Торидальная ОС имеет невысокую скорость коммуникаций (максимальная длина пути в ОС из  $n^2$  клеток равна  $n/2$ ). Реализация ОС в виде двоиного гиперкуба (см. § 5.4) размерности  $d(n=2^d)$  снижает максимальную длину пути до  $d=\ln n$ , но весьма неэффективна по количеству соединений и занимаемой площади.

В последние годы активно разрабатывались топологии ОС, удовлетворяющие в той или иной степени вышеприведенным требованиям. В качестве критериев рассматривались такие характеристики, как нормализованное среднее расстояние в однородной среде (произведение среднего расстояния в ОС на число портов каждой клетки), плотность сообщений (частное от деления произведения среднего расстояния и числа вершин на общее число связей), устойчивость к отказам (связей и модулей). Кроме того, учитывались удобство размещения в среде произвольного графа, простота и эффективность механизмов передачи сообщений, способность к расширению структуры.

К числу перспективных топологий ОС можно отнести так называемые кубовые циклы (cube-connected cycles) и отказоустойчивые архитектуры Прад-

хана [Прадхан, 1986], имеющие, кроме того, хорошие показатели и по другим критериям. В частности, они ограничены по числу портов (3—4), удобны для размещения на плоскости, обладают логарифмической оценкой величины среднего расстояния.

Структура ОС в виде гипердерева также считается перспективной для архитектуры ОС интеллектуальных систем, хотя и имеет худшие показатели оценок по приведенным выше критериям.

Режим передачи информации между модулями. Быстрота коммуникаций в ОС не менее важна, чем быстродействие модулей ОС [Frankel, 1986]. Передача информации по каналам в ОС может осуществляться путем коммутации каналов, а также коммутации сообщений.

При коммутации каналов модуль, породивший информацию, посылает короткое сообщение (вызов). Это сообщение пересылается коммуникационными процессорами модулей, связанных физическими каналами, в направлении модуля-адресата. Промежуточные модули резервируют определенные каналы связи для передачи сообщения. Когда вызов достигает адресата, коммуникационные модули коммутируют зарезервированные каналы и между отправителем и адресатом устанавливается связь через виртуальный физический канал. Все это время виртуальный канал не может резервироваться для передачи других сообщений.

При коммутации сообщений модуль-отправитель посылает сообщение (пакет) целиком. Оно распространяется коммуникационными процессорами других модулей параллельно до момента нахождения модуля-адресата.

Вопрос о том, какой из способов коммуникации лучше, в настоящее время не нашел ответа. При коммутации каналов сильно снижается параллелизм передачи сообщений в ОС вследствие запрета на использование зарезервированных каналов. При коммутации сообщений каналы связи используются лучше, но велика потеря времени на переписывание сообщения при передаче между соседями, что особенно заметно при обмене длинными сообщениями.

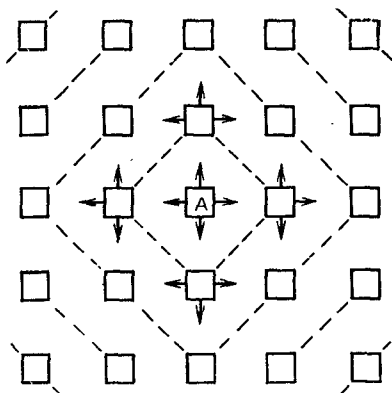
Согласно У. Хиллису, разработчику коммутационной машины (Connection Machine) необходимо соблюдать баланс между переключением цепей и пересылкой сообщений [Frankel, 1986]. Для обоих способов коммутации разработаны алгоритмы, предохраняющие от дедлоков при параллельной передаче информации. Эти алгоритмы основаны на концепциях структурированного буферного пула или виртуальных каналов.

Скорость и надежность коммуникаций в значительной степени определяется алгоритмами маршрутизации сообщений, которые, в свою очередь, существенно зависят от топологии ОС. При фиксированной маршрутизации заранее определен путь (или множество путей) прохождения сообщения между любыми модулями. При динамической маршрутизации дальнейший путь сообщения определяется коммуникационным процессором на основе локальной информации о текущей загрузке каналов связи.

Считается [Прадхан, 1986], что динамическая маршрутизация менее перспективна для ОС. Это обусловлено быстрозменяющейся загрузкой модулей и каналов связи, а также сложностью обеспечения корректности и временными затратами на реализацию алгоритма.

Одним из возможных путей организации коммутации сообщений в ОС является использование алгоритма распространения сообщений по относительному адресу [Hillis, 1985]. Метод основан на том, что в каждом модуле ОС помимо прочей информации хранятся относительные адреса клеток (модулей), которые содержат семантически связанные данные с информацией этого модуля. При необходимости передачи данных одному из модулей формируется сообщение, содержащее относительный адрес модуля-получателя. Это сообщение передается ближайшему соседнему модулю, который, не прерывая собственных вычислений, модифицирует относительный адрес и пересылает сообщение дальше до тех пор, пока оно не дойдет до получателя (в этот момент относительный адрес обнулится). В сообщении не указывается путь, которым оно должно пройти. Направление каждый раз выбирается промежуточными модулями на

Рис. 5.25. Распространение волны в ОС



основании текущего относительного адреса и локальной информации о загруженности и исправности соседних модулей. Таким образом обеспечивается высокий параллелизм при обмене информацией и устойчивость механизма передачи сообщения к задержкам к отказам.

Другим весьма перспективным путем организации обмена информацией в ОС является ассоциативный поиск, при котором используются так называемые волновые алгоритмы, работа которых состоит в следующем.

Клетка-источник высылает сообщение, в котором вместо относительного адреса содержится ассоциативный адрес либо информация, указывающая, что сообщение адресовано всем клеткам ОС. Это сообщение распространяется наподобие физической волны, согласно локальным правилам срабатывания коммутационных элементов [Гордиенко и др., 1986]. На рис. 5.25 штриховыми линиями показаны последовательные стадии распространения волны от клетки А. Пройдя по всем клеткам ОС, волна прекращает распространяться.

Волновой способ передачи информации весьма перспективен при решении задачи поиска свободной клетки в ОС, при необходимости обращения одной клетки ко многим для копирования данных или организации ассоциативного доступа к ним. Возможность максимального распараллеливания путей прохождения сообщений позволяет получить высокую скорость и надежность достижения модуля-получателя данных. Однако по сравнению с передачей сообщений по фиксированному адресу этот способ имеет недостаток — большое дублирование информации. Организация передачи сообщений этим способом неэффективна из-за участия в этом процессе коммуникационных устройств лишних модулей. Кроме того, формирование волн одновременно несколькими клетками приводит в лучшем случае к замедлению работы ОС (в худшем — к дедлокам, если не приняты специальные меры по их предотвращению). Задержки в распространении волновых сообщений могут быть также вызваны передачей длинных сообщений вследствие многократной ретрансляции отдельных ее частей.

Конструктивная и функциональная однородность модулей, возможность эффективного размещения задач с произвольной схемой их решения, выделение таких функциональных компонентов модулей, как главный и коммуникационный процессоры, память, коммутирующее устройство (индивидуальное в случае использования модулей для реализации информационного обмена по принципу близкодействия или общее для ОС с централизованным управлением от главного процессора системы [Каляев, 1984; Кодачигов, 1984], распределенность и децентрализованность обработки данных, регулярность структуры сделают ОС перспективными для создания многопроцессорных аппаратных средств поддержки механизмов распределенной обработки информации, характерных для решения интеллектуальных задач [Гордиенко и др., 1986]).

### Однородные среды для машин баз знаний

При реализации программ манипулирования со знаниями в совместных выскопроизводительных ЭВМ последовательного типа обрабатываются до сотни фактов и правил, а реальная практика создания интеллектуальных систем требует довести это количество до десятков и сотен тысяч [Hillis, 1986]. Единственным выходом из этой ситуации является распараллеливание операций над хранящимися в памяти знаниями. Эти операции, как правило, сложнее простого

нахождения строки в таблице. Нахождение нужной информации может потребовать не только просмотра всех имеющихся знаний, но и организации процедур логического вывода.

Согласно общей концептуальной схеме базы знаний [Гордиенко и др., 1985] объекты в ее различных компонентах могут быть представлены с помощью семантических сетей (СС), возможно, расширенных за счет включения определенного вида процедурных элементов.

Множество операций над знаниями [Hillis, 1985; Гордиенко и др., 1985], хранящимися в виде СС, нуждаются в эффективной аппаратной реализации. Важнейшие из них:

1) дедуктивный вывод в СС (различные принципы его осуществления) предлагались в системах представления знаний KLONE, OWL, NETL, OMEGA [Вагин, 1986].

2) сопоставление по образцу, например, на множествах продукционных правил;

3) сортировка множества объектов по определенным параметрам (например, промежуточных объектов целей по мере приближения к конечной цели);

4) поиск в графе подграфов определенной структуры.

Алгоритмы, реализующие эти операции [Вагин, 1986; Сапаты, 1986], обладают высоким потенциальным параллелизмом. При их выполнении предполагается, что нет ограничений на число модулей и на способы реализации их взаимодействия. Кроме того, запрос к БЗ порождает заранее не предсказуемые информационные обмены между хранящимися в ней объектами. Это, во-первых, указывает на неэффективность реализации этих операций на вычислительных средах, поддерживающих только регулярные информационные связи (архитектуры типа фон-неймановской; систолические, конвейерные, векторные архитектуры), а во-вторых, требует максимальной скорости коммуникации между произвольными модулями среды.

В модели ОС это требование можно выполнить, выбрав подходящий механизм коммуникации модулей. На этот выбор существенно влияет способ погружения СС в ОС. Каждая клетка-процессор в коммутационной машине Хиллиса [Hillis, 1985] хранит лишь три адреса других клеток. Поэтому каждая вершина СС преобразуется к виду сбалансированного бинарного дерева, вершины которого размещаются в соседних клетках. Количество листьев дерева определяется числом дуг, выходящих в СС из вершины и выходящих.

Клетки, соответствующие листьям деревьев, представляющих вершины семантической сети, соединяются с аналогичными клетками (листьями другого дерева) не непосредственно, а через промежуточные клетки, хранящие информацию о дугах в СС. Таким образом, объект, находящийся в клетке, не несет информацию либо о дуге СС, либо о структуре вершины СС. При таком методе размещения СС предполагается не только функциональная простота клеток, но и наличие их в очень большом количестве. Вопрос о быстрой и оптимальной по длине коммутации при условии такого размещения, не рассматривался.

В работе [Гордиенко и др., 1986] предложена реализация в ОС волновых алгоритмов дедуктивного вывода на базе специального вида СС. При волновом способе передачи сообщений скорость коммуникаций довольно высока и не зависит от способа погружения СС в ОС. Тем не менее исследуются два способа размещения СС в ОС. При первом способе в каждой клетке хранится информация о вершине СС и об инцидентных ей дугах, при втором — информация о дуге СС. Существенных преимуществ какого-либо из двух способов размещения СС в работе не отмечено.

В [Vegdahl, 1984] отмечается эффективность использования ОС для функционального программирования благодаря развитой коммуникационной структуре, возможности аппаратной реализации вычислений, эффективной организации работы с памятью.

Асинхронная реализация обработки сигналов в ОВС описывается в [Kung, 1982]. В качестве основы коммуникации используется модификация волновых алгоритмов.

Вычислительные системы на базе ОС начали выпускаться с середины 80-х годов. В 1985 г. фирма Intel выпустила машину iPSC с архитектурой, названной космическим кубом (cosmic cube), и с числом процессоров  $2^n$  ( $n=5, 6, 7$ ). Конфигурация из 128 процессоров имела быстродействие 10 Мфлоп, что составляет примерно 1/15 производительности машины Cray-1 на реальных задачах, и стоимость, в 20 раз меньшую стоимости этой машины. В iPSC использовалась коммутация каналов с аппаратной реализацией предохранения от дедлоков.

В 1986 г. фирма Thinking Machines объявила о выпуске коммутационной машины (Connection Machine). Коммутационная машина (КМ) содержит  $2^{14}$ — $2^{16}$  процессорных элементов, не реализующих операции с плавающей точкой. Массив процессоров связан с главным компьютером (типа VAX-11) через шину памяти. Поэтому ОС трактуется как «активная» память обычной машины, т. е. память произвольного доступа, над которой возможно выполнение ряда макроопераций. При их выполнении модули ОС функционируют децентрализованно. Быстродействие КМ характеризуют такие данные: подсчет частичных сумм массива из  $2^{16}$  целых чисел (200 мкс), сортировка  $2^{16}$  32-разрядных чисел (30 мс), эффективная (за несколько тактов) реализация логических и теоретических множественных операций над всей памятью. Несмотря на малую емкость внутренней памяти, каждый модуль способен поддерживать до четырех виртуальных процессоров. Таким образом, пользователь КМ получает возможность оперировать с активной памятью из  $2^{18}$  ячеек.

В 1986 г. фирма Floating Point Systems объявила о начале выпуска новой машины T-Series. Это суперЭВМ, начальная конфигурация которой состоит из четырех связанных тезерактов (четыремерных кубов с 16 вершинами), в вершинах которых находятся транспьютеры фирмы INMOS. В отличие от КМ, являющейся представителем систем с мелкозернистой структурой, T-Series содержит мощные модули — процессорные элементы производительностью до 16 Мфлоп. Быстродействие начальной конфигурации ожидается равным 1 Гфлоп.

В качестве базового языка используется новая версия языка Оккам, обогащенная новыми типами данных и расширенным множеством операций [Frankel, 1986]. С. Сейц назвал этот язык Форттрапом параллельных вычислений.

### Перспективы

В последние годы разрабатывается множество разновидностей ОС. Перед их создателями стоит ряд проблем, в частности определение функциональных возможностей и объема модулей ОС, принятие критериев оценки топологии ОС и видов маршрутизации.

Проблемой является также эффективное представление информации в ОС и разработка методологии составления алгоритмов. В самом начале своего решения находится задача создания специального языка программирования. Не решены вопросы построения операционной системы, а также ввода и вывода информации для ОС.

В настоящее время активно исследуются проблемы синхронизации и координации работы модулей ОС. Большое внимание уделяется повышению отказоустойчивости и надежности ОС. Серьезным вопросом для исследований является создание модели ОС, позволяющей произвести оценку динамических характеристик ее функционирования.

В последнее время повысился интерес к исследованиям в области построения специализированных процессоров на базе RISC-архитектуры (процессоров с сокращенным набором команд). Однородные структуры, построенные из таких процессоров, считаются весьма перспективными в качестве аппаратной поддержки систем распределенной обработки информации и интеллектуальных систем.



## 5.4. Специализированные вычислительные структуры

А. П. Горяшко

### Основные положения

Главная трудность при реализации почти любой прикладной интеллектуальной системы состоит в выборе и построении высокопроизводительной вычислительной структуры. Недостаточная «вычислительная мощность» не только традиционных однопроцессорных ЭВМ фон-неймановской архитектуры, но и параллельных ЭВМ типа SIMD (таких, например, как STAR-100, CYBER-205, CRAY) очевидна и связана с тем, что подобные ЭВМ ориентированы на определенные классы задач вычислительной математики и именно на этих классах достигают предельных возможностей по производительности. Интеллектуальные задачи имеют, как правило, совершенно иную вычислительную природу и для них необходимо создание специализированных вычислительных сред.

Создание таких сред предполагает прежде всего изучение специфических требований, предъявляемых задачами в области ИИ к архитектуре вычислительных сред, и согласование этих требований с возможностями и ограничениями интегральной технологии.

Предполагается, что при построении любой вычислительной структуры возможен пропорциональный размер оборудования на время решения, т. е. возможно создание специализированной структуры из  $N$  процессоров, которая решает задачу в  $N$  раз быстрее, чем один такой же процессор.

Этот эффект может быть достигнут, только когда каждый процессор структуры в каждый момент времени получает необходимые ему данные, т. е. минимальны простои процессоров в структуре. Для этого необходимо или создавать максимально специализированную структуру, идеально подстроенную под конкретную задачу, или предусмотреть такую структуру межсоединений, которая позволяет максимально быстро связать любую пару элементов структуры. Между «предельно специализированной» и «предельно универсальной» структурами межсоединений имеет место обширная область промежуточных решений — классов вычислительных структур с различными характеристиками сложности, стоимости, отказоустойчивости и т. п.

Интенсивные исследования (и не только теоретические), предпринятые в последние 10 лет, позволяют классифицировать предлагаемые структуры с точки зрения различных свойств: топологических, надежных, технологических.

При анализе топологических свойств структуры исследуются вопросы, связанные с возможностью «общения» элементов структуры между собой в процессе обработки информации. Например, в результате подобного анализа оценивается сложность (в числе элементарных шагов) передачи информации от любого элемента к любому другому (или наоборот удаленному), средняя нагрузка элементов структуры и т. п. При анализе надежных свойств необходимо понять трудоемкость процедур обнаружения неисправностей в структуре; последствия (с точки зрения целостности функционирования) возникновения неисправностей; сложность передачи сообщений при наличии неисправных элементов; возможности реконфигурации структуры с неисправными элементами для организации «мягкой деградации».

Технологические свойства оценивают в первую очередь эффективность и возможность интегральной реализации структуры. Это регулярность и однородность структуры, площадь, занимаемая структурой в прямоугольной решетке, максимальная длина связей и т. п.

Каждый топологически различный класс структур характеризуется своим набором свойств, и строго упорядочить эти структуры без анализа алгоритмической структуры задач, для решения которых они предназначены, невозможно. Поэтому далее будет приведен анализ основных «базовых» топологических структур и их модификаций, а также даны соображения о перспективности выбора той или иной структуры для некоторых классов задач ИИ.

Интерес к изучению вычислительных возможностей регулярных структур из большого числа одинаковых автоматов сформировался практически одновременно с основными исследованиями в области теории автоматов. Понятие однородной вычислительной среды (клеточная или сотообразная структура) впервые введено в лекциях Дж. Неймана, прочитанных в 1949 г.

В этой работе, а затем в работах Э. Мура [Мур, 1966] и А. Беркса [Burks, 1960] анализировались в основном двумерные неограниченные однородные сети автоматов (которые впоследствии получили название автоматы Неймана — Черча) с точки зрения возможности самовоспроизведения первоначально заданных конфигураций состояний элементов. В частности, было выяснено наличие недо-стижимых конфигураций (конфигурации «райского сада»).

Затем наиболее серьезные результаты в этой области исследований появляются в СССР. В цикле работ Я. М. Барздиня [Барздинь, 1965] исследовались автоматные модели, более общие, нежели автоматы Неймана — Черча. Рассмотрены модели вычислительных средств, представленных ориентированными мультиграфами с симметричными (двунаправленными) ребрами. Введено понятие емкости такой среды  $G$ , как функция  $f^G = \max f_a^G(r)$ , где  $f_a^G(r)$  — число тех вершин, которые находятся на расстоянии, не большем чем  $r$  от  $a$ , а максимум берется по всем вершинам  $a$  среды  $G$ . Показано также [Барздинь, 1966], как меняется число элементов и скорость вычисления при моделировании автоматов с одной емкостью на автоматах с другой емкостью. Связь между емкостью различных автоматных сред (деревья, гиперкубы, решетки) и вероятностными оценками времени передачи информации в таких средах изучались в [Го-ряшко, 1972].

Окончательные (с точностью до порядка величины) оценки были установлены при изучении моделирования поведения произвольной логической сети, содержащей  $N$  элементов, в двумерной решетке автоматов Неймана — Черча [Барздинь, 1966]. Выяснено, что поведение любой логической сети из  $N$  элементов может быть смоделировано в двумерной решетке, содержащей  $O(N \log N)$  автоматов. Эти результаты затем развиты в работах [Подколзин, 1975; Аль-брехт, 1978].

Подробно были изучены вопросы реализации логических сетей, вычисляющие произвольные функции алгебры логики (ФАЛ) в двумерных решетках. Здесь получено два типа результатов: о поведении длины связей в двумерных решетках [Коршунов, 1967] и о плотности размещения схем, реализующих произвольные ФАЛ  $f_n$  в двумерных решетках [Глазунов и др., 1968]. В первом случае доказано, что, если при реализации  $f_n$  схемой  $S_n$  длины связей ограничены не зависящей от  $n$  константой, число элементов в схеме  $S_n$  будет  $L(S_n) = 2^n$ , т. е. на порядок больше, чем при отсутствии ограничений на длину. Аналогично при условии, что схема  $S_n$  должна располагаться «плотно» (отношение числа занятых логическими элементами узлов решетки ко всем узлам не меньше некоторой не зависящей от  $n$  константы), число элементов в схеме оказывается не меньше чем  $O(2^n)$ . Все попытки разместить в двумерной решетке оптимальные по порядку числа элементов схемы, реализующие  $f_n$ , приводят для почти всех  $f_n$  к возрастанию площади до величины  $(O(2^n/n^2))^2$ . Здесь напрашивается сравнение с окончательными по порядку оценками площади для сетей межсоединений (multistage interconnection networks), содержащими  $N \log N$  элементов и занимающими площадь  $(N/\log N)^2$  [Kleitman et al., 1981].

Офманом [Офман, 1965] впервые рассмотрены вычислительные возможности структуры, содержащей, как теперь принято говорить, «сеть межсоединений», которая осуществляет любую подстановку из  $N$  в  $N$ .

С конца 70-х годов в СССР практически прекращаются публикации теоретических работ, относящихся к изучению процессорных структур, отличных от «решеток». Одновременно резко возрастает интерес к этой проблеме за рубежом, что объясняется в первую очередь успехами интегральной технологии. Уже на конференции, организованной в 1979 г. Калифорнийским технологиче-

ским институтом «Архитектура, конструирование и изготовление БИС» [Rem, 1979], обсуждались общесистемные вопросы проектирования однокристалльных устройств сложностью порядка 10 млн. активных элементов. Предполагалось, что подобные кристаллы появятся в начале 90-х годов и теория к тому времени должна подсказать, что именно и как следует реализовать на кристалле такой сложности. В 80-е годы сформировалось три теоретических направления: исследование вычислительных возможностей топологически разных процессорных структур; теоретико-сложностная проблематика проектирования СВИС; исследование возможностей построения отказоустойчивых структур. Одновременно идет поиск классов задач, решение которых с наибольшей эффективностью возможно в специализированных процессорных структурах, реализованных в виде СВИС или на их базе. К ним относятся задачи из области ИИ: распознавание изображений, моделирование процессов восприятия, моделирование познавательной деятельности, логический вывод, поиск информации в семантических сетях, построение экспертных систем на базе систем продукции. Состоянию дел в этой области посвящен, в частности, специальный номер Computer, 1987, № 1, где перечислены как реализованные, так и задуманные проекты интеллектуальных машин для производственных систем и семантических сетей; объектно-ориентированные машины, а также машины интеллектуального интерфейса.

### Основные определения

Моделью элемента вычислительной структуры может служить конечный автомат, описываемый как пятерка  $\langle X^n, Y^m, Q, \Psi, \Phi \rangle$ , где  $X^n$  — входной алфавит,  $Y^m$  — выходной алфавит,  $Q$  — внутренний алфавит,  $\Psi$  — функция переходов (отображение  $X^n \times Q$  в  $Q$ ) и  $\Phi$  — функция выходов (отображение  $X^n \times Q$  в  $Y^m$ ).

Различают два вида элементов ВС — *основные* (П-элементы) и *переключа-тельные* (л-элементы). К числу основных относят процессорные модули и модули памяти. Переключаательные элементы осуществляют заданное соединение входных полюсов с выходными и в отличие от основных имеют специфические особенности.

Самый простой л-элемент — это элемент с двумя входами ( $x_1, x_2$ ) и двумя выходами ( $y_1, y_2$ ). Такой элемент может иметь не более 16 состояний: каждый вход может находиться в четырех состояниях (отсутствие соединения, соединение с выходом  $y_1$  и  $y_2$  одновременно).

Таблица 16 состояний показана на рис. 5.26. л-элементы, имеющие только два состояния ( $S_6$  и  $S_{10}$ ), называются л-элементами 1-го типа (селекторами); имеющие четыре состояния ( $S_3, S_5, S_{10}, S_{12}$ ) — л-элементами 2-го типа и имеющие шесть состояний ( $S_1, S_2, S_4, S_5, S_8, S_{10}$ ) — л-элементами 3-го типа (banуap).

Более общий тип л-элемента — это элемент  $a \times b$ , с  $a$  входами и  $b$  выходами, причем из  $a$  входов независимо от остальных входов может быть соединение с любым (но только одним) выходом. Это л-элемент 4-го типа (cross-bar). л-элементы 1-го типа удобно представлять графовой моделью так, как показано на рис. 5.27 [Agrawal, 1983].

**Классы вычислительных структур (ВС).** Два основных класса ВС — это *структуры межсоединений* (multistage interconnection network (MIN)) и *процессорные структуры* (computer network).

В классе структур межсоединений (СМ) рассматриваются сети, образованные л-элементами. Задача такой сети — осуществить заданное соединение входных П-элементов (например, процессорных модулей) с выходными П-элементами (например, модулями памяти). В классе процессорных структур рассматриваются сети, образованные П-элементами, т. е. сети в базе из П-элементов. Задача такой сети — реализация некоторого класса алгоритмов обработки информации при заданных ограничениях на время обработки.

Класс структур, объединяющий упомянутые выше структуры, называется классом перестраиваемых процессорных структур (ППС). В классе ППС рассматриваются сети в базе из л- и П-элементов. Основные задачи класса ППС:

$S_0$		$S_8$	
$S_1$		$S_9$	
$S_2$		$S_{10}$	
$S_3$		$S_{11}$	
$S_4$		$S_{12}$	
$S_5$		$S_{13}$	
$S_6$		$S_{14}$	
$S_7$		$S_{15}$	

Рис. 5.26. Таблица состояний для элемента с двумя входами и двумя выходами

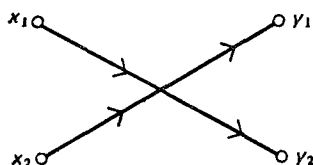


Рис. 5.27. Графовая модель  $\pi$ -элемента первого типа

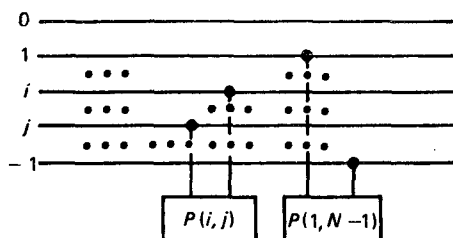


Рис. 5.28. Подсоединение элементов к шинам

реализация заданных алгоритмов обработки информации при неисправностях  $\Pi$ -элементов за счет функционального исключения неисправных  $\Pi$ -элементов из процесса обработки информации в сети;

реализация в одной сети различных графов обработки информации за счет изменения топологии сети.

Любой из классов ВС может быть задан графом  $G = \{V, E\}$ . Здесь  $V$  — множество вершин от 0 до  $N-1$ , а  $E$  — множество ребер  $(i, j)$ , где ребро  $(i, j)$  соединяет вершины с номерами  $i$  и  $j$  (наличие ориентации в графе оговаривается особо).

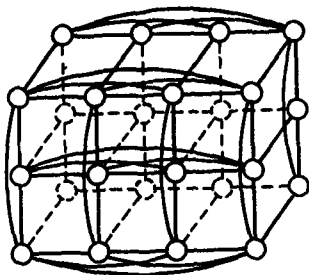
Множество вершин  $V$  содержательно может интерпретироваться как множество элементов базиса ( $\Pi$ - или  $\pi$ -элементов) или как множество двунаправленных связей (шины) между элементами.

В первом случае принято говорить об архитектуре связей, обозначая  $LA(G) = \{P, G\}$ , где  $P$  — множество, содержащее основные и/или переключательные элементы, а  $G$  — множество двунаправленных связей между этими элементами.

Во втором случае говорят о шинной архитектуре  $BA(G)$ , обозначая  $BA(G) = \{B, P\}$ . Здесь  $B$  — множество шин с нумерацией  $B(0), B(1), \dots, B(N-1)$ , соответствующих  $N$  вершинам графа, а  $P$  — множество элементов базиса, причем  $P(i, j)$  означает, что элемент базиса присоединен к шинам с номерами  $i, j$  (рис. 5.28).

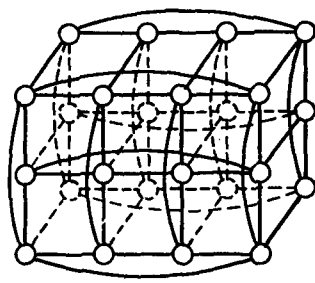
Далее везде предполагается, что граф ( $LA(G)$  или  $BA(G)$ ) имеет ровно  $N$  вершин.

Для топологически различных графов  $G$ , задающих процессорные структуры, введем следующие обозначения.



$C_1(4, 2, 3)$

Рис. 5.29. Обобщенный гиперкуб



$C_1'(4, 2, 3)$

Рис. 5.30. Обобщенный кольцевой гиперкуб

1. Полный граф  $G_0$ : каждая вершина  $v$  соединена ребром с каждой из  $(N-1)$ -й вершиной.

2. Кольцо  $R_0$ : каждая вершина  $v$  ( $0 \leq v \leq N-1$ ) соединена ребром с вершинами  $v+1 \pmod N$  и  $v-1 \pmod N$ .

3. Хордовое кольцо  $R_1(s)$ : каждая вершина  $v$  ( $0 \leq v \leq N-1$ ) соединена дугой с вершиной  $v+1 \pmod N$  и дугой с вершиной  $v-s \pmod N$  ( $s \geq 2$ ) [Raghavendra, Gerla, Avizienis, 1985].

4. Двоичный гиперкуб размерности  $n$  ( $N=2^n$ ): каждая вершина  $v$  задана двоичным числом  $p(v) = (p_0 \dots p_i \dots p_{n-1})$  и ребрами соединены те и только те пары вершин  $v, u$ , для которых  $\bar{p}(v) \oplus \bar{p}(u) = 1$ , где  $p(v) \oplus \bar{p}(u)$  обозначает операцию покомпонентного суммирования по  $\pmod 2$   $n$ -мерных двоичных векторов; можно также определить  $C_0(n)$  с помощью  $n$  функций вида

$$C_i(p_0 \dots p_{i-1} p_i p_{i+1} \dots p_{n-1}) = p_0 \dots p_{i-1} \bar{p}_i p_{i+1} \dots p_{n-1}.$$

5. Обобщенный гиперкуб размерности  $n$   $C_1(m_1, m_2, \dots, m_n)$  ( $N=m_1 \times m_2 \times \dots \times m_n$ ). Для всех  $i=1, \dots, n$ ,  $m_i > 1$  каждая вершина  $v$  задана числом со смешанным основанием  $r(v) = (x_1 \dots x_i \dots x_n)$ , где  $0 \leq x_i \leq m_i - 1$ , и вершина с номером  $(x_1 \dots x_{i-1} x_i x_{i+1} \dots x_n)$  соединена ребром с теми и только теми вершинами, номера которых  $(x_1 \dots x_{i-1} \tilde{x}_i x_{i+1} \dots x_n)$  для всех  $1 \leq i \leq n$ , где  $\tilde{x}_i$  пробегает все целые значения между 0 и  $m_i - 1$ , за исключением  $x_i$ . Таким образом, обобщенный гиперкуб — это  $n$ -мерный гиперкуб с  $m_i$  вершинами в каждом измерении (рис. 5.29) [Bhuyan et al., 1984].

6. Обобщенный кольцевой гиперкуб размерности  $n$   $C_1'(m_1, m_2, \dots, m_n)$  ( $N=m_1 \times m_2 \times \dots \times m_n$ ). Вершины заданы так же, как и в случае  $C_1(m_1, m_2, \dots, m_n)$ , но каждая вершина с номером  $(x_1 \dots x_{i-1} x_i x_{i+1} \dots x_n)$  соединена ребром с вершинами

$$(x_1 \dots x_{i-1} (x_i + 1) \pmod{m_i} x_{i+1} \dots x_n) \quad \text{и} \quad (x_1 \dots x_{i-1} (x_i - 1) \pmod{m_i} x_{i+1} \dots x_n)$$

для всех  $1 \leq i \leq n$  (рис. 5.30) [Bhuyan, Agrawal, 1984].

7. Кубосвязный цикл размерности  $s$   $C_2(h, s)$ , ( $h \geq s$ ,  $N=h \cdot 2^s$ ) задается с помощью  $C_0(s)$  заменой каждой вершины  $v$  ( $v=0, \dots, 2^s-1$ ) кольцом  $R_0$  из  $h$  вершин, причем к каждой вершине кольца с номерами  $0, 1, \dots, s$  присоединено одно и только одно ребро из  $s$  ребер, связанных с вершиной  $v$  в кубе  $C_0(s)$ . Таким образом,  $s$  вершин в каждом из колец  $R_0^i$  ( $i=0, \dots, 2^s-1$  имеют степень 3, а  $h-s$  вершин — степень 2 (рис. 5.31) [Valeeje, Kuo, Fuchs, 1986].

8. Бинарное дерево глубины  $n$   $T_0(n)$  ( $N=2^n-1$ ) содержит  $n$  рангов  $r$  ( $r=0, \dots, n-1$ ), причем в каждом ранге размещается ровно  $2^r$  вершин так, что каждая вершина ранга  $r$  ( $0 \leq r \leq n-2$ ) соединена со своей парой вершин ранга  $r+1$ . Таким образом, каждая вершина в рангах  $1, \dots, n-2$  имеет ровно одно входящее и два исходящих ребра, вершина ранга 0 (корень) имеет 2 исходящих

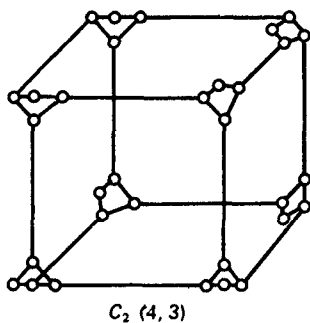


Рис. 5.31. Кубосвязный цикл

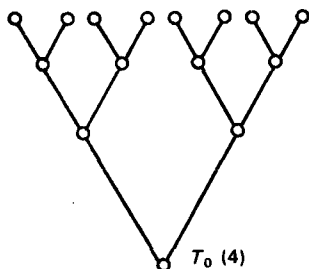


Рис. 5.32. Бинарное дерево

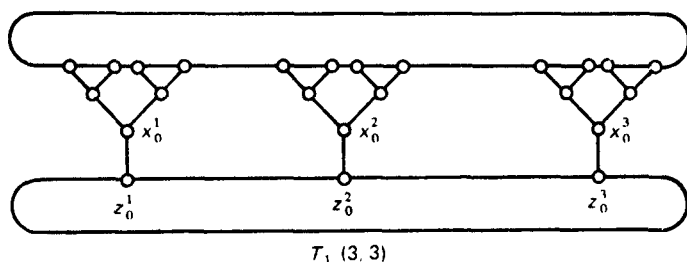


Рис. 5.33. Мультидерево

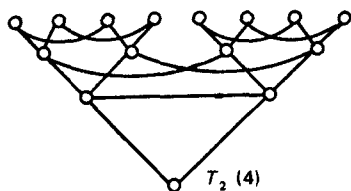


Рис. 5.34. Гипердерево

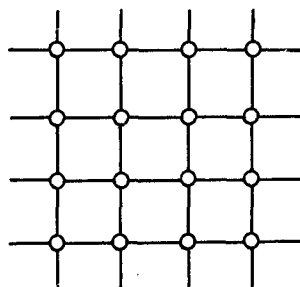


Рис. 5.35. Двумерная решетка

ребра и все вершины ранга  $n-1$  (крона) — по одному входящему ребру (рис. 5.32).

9. Мультидерево глубины  $n$  и ширины  $k$   $T_1(n, k)$  ( $N = k \cdot 2^n$ ) содержит  $k$  двоичных деревьев  $T_0(n)$ , объединенных с помощью колец  $R_0$ . Корневая вершина  $x_0^i$  ( $i=1, \dots, k$ ) каждого из  $k$  деревьев соединена ребром с вершиной  $z_0^i$ , а вершины  $z_0^1, \dots, z_0^k$  образуют кольцо  $R_0$ . Второе кольцо образовано вершинами кроны всех  $k$  деревьев и, следовательно, содержит  $k2^{n-1}$  вершин (рис. 5.33).

10. Гипердерево глубины  $n$   $T_2(n)$  ( $N = 2^n - 1$ ) отличается от бинарного дерева  $T_0(n)$  наличием у каждой вершины (кроме корня) дополнительного ребра, соединяющего пары вершин одного ранга. Вершина  $v$  каждого ранга  $r$  задана

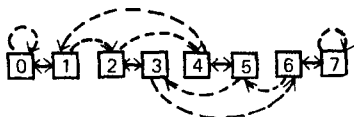


Рис. 5.36. Соединение тасовки-обмена:

↔ обмен, - - -> тасовка

двоичным числом  $\bar{p}(v) = p_0 \dots p_{r+1}$ . Пары вершин  $v, u$ , для которых  $\bar{p}(v) \oplus \bar{p}(u) = 1$ , соединены ребрами (рис. 5.34) [Haynes et al., 1982].

11. Двумерная решетка ( $N=2^n$ )  $S_2(n)$ . Каждая вершина  $v$  решетки задается декартовыми координатами  $(x, y)$ , где  $1 \leq x \leq n, 1 \leq y \leq n$ , и соединена ребрами с ближайшими соседями, т. е. с вершинами, координаты которых  $(x+1, y), (x-1, y), (x, y+1), (x, y-1)$  (рис. 5.35).

12. Соединение тасовки-обмена SE (shuffle-exchange). Каждая из  $N$  вершин  $v$  задана двоичным числом  $p(v) = (p_0 p_1 \dots p_{n-2} p_{n-1})$ . Соединение любой вершины задается парой функций: функцией тасовки влево  $t'(p_0 p_1 \dots p_{n-2} p_{n-1}) = p_{n-1} p_0 p_1 \dots p_{n-2} p_{n-2}$  или вправо  $t''(p_0 p_1 \dots p_{n-2} p_{n-1}) = p_1 \dots p_{n-2} p_{n-1} p_0$  и функцией обмена  $O(p_0 p_1 \dots p_{n-2} p_{n-1}) = p_0 p_1 \dots p_{n-2} p_{n-1}$ , т. е. вершина с номером  $(p_0 p_1 \dots p_{n-2} p_{n-1})$  соединена с вершинами, номера которых определяют функции  $t$  и  $O$  (рис. 5.36) [Siegel, 1979].

13. Соединение  $\ll n \pm 2^i \gg PM(i)$ . Вершина с номером  $j$  ( $j=1, \dots, N$ ) соединена дугой с вершиной, номер которой  $(j+2^i) \bmod N$ , и с вершиной, номер которой  $(j-2^i) \bmod N$  (рис. 5.37).

14. Соединение обобщенной тасовки-обмена (сети Pradhan)  $P(n, m)$  ( $N = m^n$ ); каждая вершина  $v$  задана числом со смешанным основанием  $p(v) = (x_0 \dots x_{n-1})$ , где  $0 \leq x_i \leq m-1$ . Граф  $P(n, m)$  строится в два этапа. На первом этапе строится скелетный граф  $P'(n, m)$  с помощью операций обобщенной тасовки-обмена, т. е. вершина графа  $P'(n, m)$  с номером  $\bar{p}(v)$  соединена с такими вершинами, номера которых определяются функциями  $\bar{t}(x_0, \dots, x_{n-1}) = x_1 x_2 \dots x_{n-1} x_0$  и  $\bar{O}(x_0, \dots, x_{n-1}) = * x_1 x_2 \dots x_{n-1}$ , где  $*$  — произвольное значение, отличное от  $x_0$ . (Очевидно, при  $m=2$  функция  $\bar{t}(\cdot)$  — это тасовка вправо, а функция  $\bar{O}(\cdot)$  — функция обмена.)

На втором этапе скелетный граф  $P'(n, m)$  достраивается по правилам, которые показаны на примере случая  $n=2$ . Пусть  $k = (m^n - 1)/(m - 1)$ . Тогда при  $n=2$  и  $m$  четном ребрами соединяются вершины с номерами  $(0, k); (2k, 3k); \dots; ((m-2)k, (m-1)k)$ , т. е. всего  $m/2$  дополнительных ребер. При  $n=2$  и  $m$  нечетном к графу  $P'(n, m)$  добавляется дополнительная вершина  $u$ , которая соединяется ребрами с вершинами  $0, k, 2k, \dots, (m-1)k$ , т. е. всего  $m$  дополнительных ребер.

Для построенного таким образом графа  $P(2, m)$  степень любой вершины равна  $m$ . В случае, когда  $n \geq 3$ , степень любой вершины  $m+1$ .

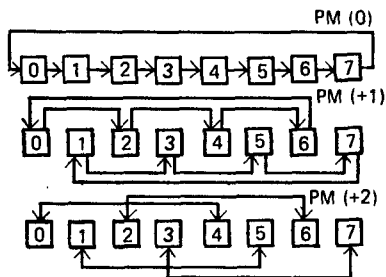


Рис. 5.37. Соединение  $\ll n \pm 2^i \gg$

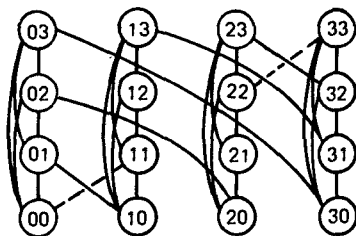


Рис. 5.38. Соединение обобщенной тасовки-обмена

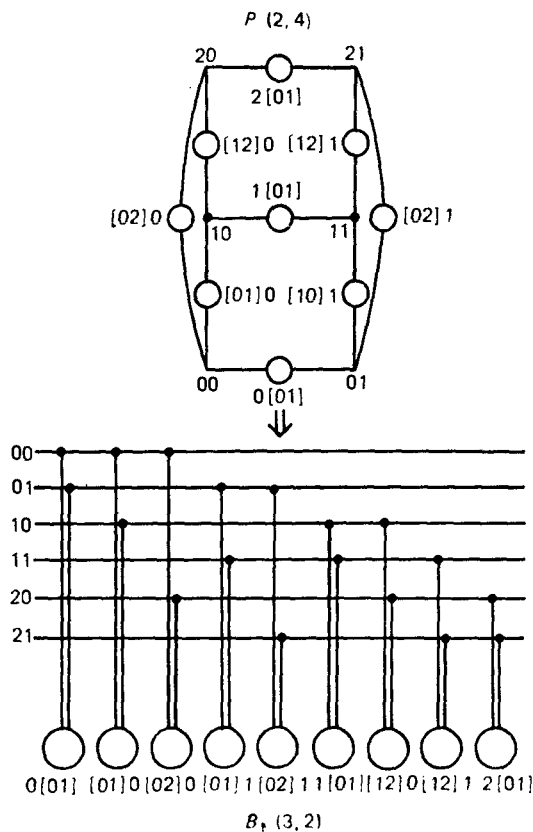


Рис. 5.39. Обобщенная гипершина

На рис. 5.38 приведен граф  $P(2, 4)$ , где каждая вершина задана двухразрядным числом по основанию 4. Штриховыми линиями показаны дополнительные ребра [Pradhan, 1985].

15. Обобщенная гипершина  $B_1(m_1, m_2, \dots, m_n)$  размерности  $(N = m_1 \times m_2 \dots \times m_n)$ . Представитель линейной архитектуры — полный аналог структуры  $C_1(m_1, m_2, \dots, m_n)$ . Каждая вершина (соответствующая шине) задана числом со смешанным основанием  $(x_1, x_2, \dots, x_i, \dots, x_n)$ , где  $0 \leq x_i \leq m_i - 1$ . Ребро (соответствующее элементу базиса) задается числом  $(x_1, \dots, x_{i-1}, [z_i y_i] x_{i+1}, \dots, x_n)$ , где  $y_i, z_i \in [0, 1, \dots, m_i - 1]$ , что означает подсоединение элемента базиса к двум шинам  $(x_1, \dots, x_{i-1} y_i x_{i+1}, \dots, x_n)$  и  $(x_1, \dots, x_{i-1} z_i y_i x_{i+1}, \dots, x_n)$  (рис. 5.39) [Pradhan, 1985].

Определим теперь некоторые структуры межсоединений, каждая из которых содержит  $N$  входных и  $N$  выходных полюсов ( $N = 2^n$ ).

16. Межсоединения на основе тасовки-обмена  $SCM(t)$  (Shuffle CM) содержат  $\log_2 N$  рангов, в каждом из которых размещено  $N/2$   $t$ -элементов типа  $t$  ( $t = 1, \dots, 3$ ), и элементы ранга  $i$  соединены ребрами только с элементами ранга  $i+1$  ( $1 \leq i \leq (\log_2 N) - 1$ ).

Соединение  $N$  выходных полюсов ранга  $i$  с  $N$  входными полюсами ранга  $i+1$  задается с помощью соединения тасовки влево или тасовки вправо. Это



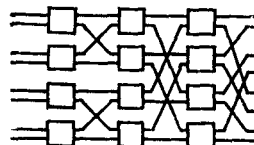
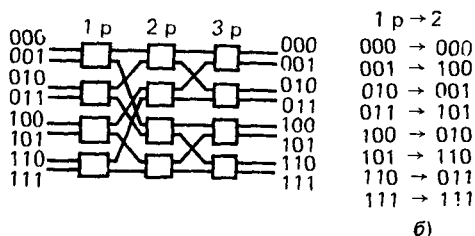


Рис. 5.40. Межсоединение на основе тасовки-обмена влево:

а — схема, б — пример тасовки между первым и вторым рангом

Рис. 5.41. Межсоединение на основе тасовки-обмена вправо

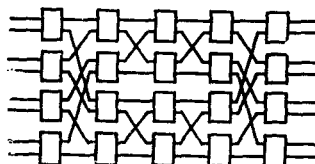


Рис. 5.42. Сеть Бенеша

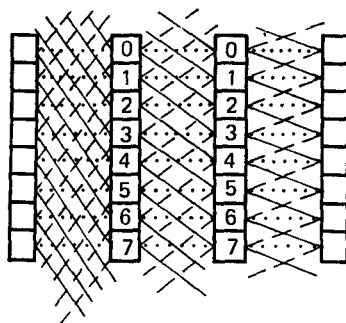


Рис. 5.43. Расширенная сеть межсоединений

означает, что каждый из  $N$  полюсов  $v$  задается двоичным числом  $\tilde{p}(v) = (p_0 p_1 \dots p_{n-2} p_{n-1})$ . В первом ранге выходные полюса нумеруются в естественном порядке 1, 2, ...,  $N$ . Порядок нумерации входных полюсов элементов 2-го ранга определяется функцией  $t'$  ( $t''$ ), примененной к двоичным номерам первого ранга. Полюса с одинаковыми номерами соединяются ребрами. В следующем ранге соответствующая функция тасовки применена к двоичным номерам предыдущего ранга и т. д. Последовательное применение  $n$  раз функции тасовки ( $t'$  или  $t''$ ) приводит к первоначальной нумерации (рис. 5.40,а, случай  $t'$ , рис. 5.41, случай  $t''$ ). На рис. 5.40,б показан пример тасовки влево между первым и вторым рангом сети [Agrawal, 1983].

17. Сеть Бенеша BCM: содержит  $2 \log_2 N - 1$  ранг, в каждом из которых размещено  $N/2$   $n$ -элементов типа 1 (селектор). Первые  $n$  рангов этой сети совпадают с сетью SCM (1), соединения в ранге  $n+i$  ( $i=1, \dots, n-1$ ) совпадают с соединениями в ранге  $n-i+1$  (рис. 5.42) [Agrawal, 1983].

18. Расширенная сеть межсоединений ACM (Augmented CM) содержит  $\log N$  рангов, в каждом из которых  $N$   $n$ -элементов типа 4 с тремя входами и тремя выходами. Соединения элементов в каждом ранге описываются как  $PM(i)$ , а именно один выход элемента с номером  $x$  в ранге  $i$  соединен со входом элемента с номером  $(x+2^i) \bmod N$  в ранге  $i+1$  и третий выход — со входом элемента с номером  $(x-2^i) \bmod N$  в ранге  $i+1$  (рис. 5.43) [Siegel, 1979].

**Параметры процессорных структур и структур межсоединений.** Расстоянием  $d(u, v)$  между вершинами  $u$  и  $v$  графа  $G$  называется длина кратчайшего

пути, соединяющего их (если  $u$  и  $v$  не соединены  $d(u, v) = \infty$ ). В связном графе расстояние является метрикой.

Диаметр  $D(G)$  связного графа  $G$

$$D(G) = \max_{u, v} d(u, v),$$

где максимум берется по всем парам вершин графа  $G$ .

Степенью  $s$  вершины  $v$  в графе  $G$  называется число ребер, инцидентных  $v$ . Граф регулярен, если все вершины имеют одинаковую степень.

Реберной связностью  $\lambda(G)$  графа  $G$  называется наименьшее число ребер, удаление которых приводит к несвязному графу

$$\lambda(G) \leq s(G).$$

Среднее расстояние  $\bar{d}$  в графе  $G$ , содержащем  $N$  вершин, определяется как

$$\bar{d} = \sum_{d=1}^D dN_d / (N - 1),$$

где  $N_d$  — число вершин, находящихся на расстоянии  $d$  от вершины, выбранной в качестве начальной.

Средняя плотность движения  $\rho$  в графе  $G$ , содержащем  $N$  вершин:

$$\rho = \bar{d}N/Z,$$

где  $Z$  — общее число ребер в графе. Для регулярного графа  $Z = Ns/2$  и, значит,  $\rho = 2s\bar{d}$ .

Рассматривают два вида предположений о возможных неисправностях в ПС и СМ. Первое предположение — возможно наличие любых  $k$  неисправностей типа константа 0, константа 1 на входах и выходах всех элементов базиса, в котором реализована сеть. Здесь  $k$  — кратность неисправности.

Второй тип предположений (так называемые нормальные предположения теории отказоустойчивости):

- любая связь и любой элемент могут быть неисправны;
- неисправный компонент недоступен;
- неисправности независимы.

Нормальные предположения теории отказоустойчивости для графовой модели означают разрыв (удаление) некоторого множества ребер в рассматриваемом графе<sup>1</sup>.

Отказоустойчивостью структуры, представленной графом  $G$  (обозначение  $FT(G)$ ), принято называть максимальное число ребер, удаление которых еще оставляет граф  $G$  реберно-связным. Таким образом, для регулярного графа  $G$

$$FT(G) = s(G) - 1.$$

**Маршрутизация (навигация)** — способ передачи сообщений от  $\Pi$ -элемента — источника к  $\Pi$ -элементу — адресату.

**Трансляция** — маршрутизация, при которой источник передает сообщение всем  $\Pi$ -элементам в сети. Маршрутизация называется адаптивной в том случае, когда каждый исправный  $\Pi$ -элемент имеет список номеров всех неисправных  $\Pi$ -элементов.

Определения, которые приводятся ниже, относятся только к структурам межсоединений.

Взаимнооднозначное отображение конечного множества элементов в себя называется *подстановкой*. Подстановка является *допустимой* в СМ, если су-

<sup>1</sup> Модели неисправностей, рассмотренные здесь, конечно, не описывают все возможные физические неисправности реальных схем. Их основное достоинство в возможности точно сформулировать задачу и получить легко интерпретируемые результаты.

# Оценки параметров

Тип структуры	Диаметр $D$	Степень $s$	Стоимость $C = sD$
$G_0$	1	$N - 1$	$N - 1$
$R_0$	$N/2$	2	$N$
$R_1(u)$	$\left\lceil \frac{N}{u+1} \right\rceil + (u-1)$	3	$\min 6 \sqrt{N}$ при $u = \lceil \sqrt{N} \rceil$
$C_2(n)$	$\log_2 N$	$\log_2 N$	$\log_2^2 N$
$C_1(m_1, \dots, m_n)$	$n$	$1,5 \log_2 N$	$0,75 \log_2^2 N$
$C_1^r(m_1, \dots, m_n)$	$\sum_{i=1}^n \lceil m_i/2 \rceil$	$2n$	$0,889 \log_2^2 N$ при $n = \lceil \log_2 N \rceil$
$C_2(h, s)$	$\approx \log_2 N$	3	$\approx 3 \log_2 N$
$T_0(n)$	$\sim 2 \log_2 N$	3	$6 \log_2 N$
$T_1(n, k)$	$\sim 2 \log_2 \frac{N}{k}$	3	$\sim 6 \log_2 N/k$
$T_2(n)$		4	
$B_1(m_1, \dots, m_n)$	$n + 1$	$\sum_{i=1}^n m_i - 1$	$\approx m \log_m^2 N$ при $N = m^n$
$P(m, n)$ $N = m^n$	$2n - 1$	$m$ или $m + 1$	$2m \log_m N$

## процессорных структур

Среднее расстояние $\bar{d} = \sum_{r=1}^D r N_r / N - 1$	Средняя плотность сообщения $\rho = \bar{d} N / Z$	Общее число связей $Z$	Отказоустойчивость $F$	Площадь $A$
1	$(N - 1)^{-1}$	$N(N - 1)$	$N - 1$	$\sim N^4$
$(N + 1)/4$	$(N + 2)/4$	$N$	1	$2N$
$\frac{N + u^2}{4u}$ $\min \frac{1}{2} \sqrt{N}$	$\min \frac{1}{4} \sqrt{N}$	$2N$	2	$\sim 2N$
$\approx 0,5 \log_2 N$	$\sim 1$	$0,5N \log_2 N$	$(\log_2 N) - 1$	
$0,375 \log_2 N$	$\sim 0,5$	$\frac{N}{2} \sum_{i=1}^n (m_i - 1)$	$(1,5 \log_2 N) - 1$	
$0,667 \log_2 N$ при $m_i = 8$ для всех $i$	2	$0,334N \log N$	$(0,667 \log_2 N) - 1$	
$\approx 0,8 \log_2 N$	$\sim 1,6$	$hN + 0,5N \log_2 N$	2	$O\left(\left(\frac{N}{\log N}\right)^2\right)$
$\sum_{i=1}^{n-1} i 2^i$ $\frac{i-1}{N-1} \sim \log N$	$\log N$	$N$	1	$2N$
		$\frac{3N}{2}$	2	$O(N)$
$\approx 1,1 \log N$	$0,55 \log N$	$\approx 2N$	2	$\sim 12N$
	0,5	$N$	2	
		$0,5Nm$	$m - 1$ или $m$	$O\left(\left(\frac{N}{\log N}\right)^2\right)$

Тип структуры	Диаметр $D$	Степень $s$	Стоимость $C = sD$
$S_2$	$2\sqrt{N}$	4	$8\sqrt{N}$
SCM (1)	$\sim \log_2 N$	4	$\sim 4 \log_2 N$

существует такое состояние СМ, которое реализует эту подстановку. Сеть межсоединений, в которой элементы соединены между собой так, что любой входной полюс сети может быть соединен с любым, но только одним выходным полюсом, называется полноступенчатой.

В некоторых типах СМ любой входной полюс может быть соединен с любым выходным более чем одним путем. В этом случае принято говорить об альтернативных путях. Так, в АСМ каждый вход соединен с каждым выходом тремя альтернативными путями. В ВСМ существует  $N/2$  альтернативных путей между любым входом и любым выходом.

При попытке реализовать в СМ заданную подстановку возникает конфликт, если оказывается необходимым установить какой-либо из  $n$ -элементов в состояние  $S_6$  или  $S_9$ , т. е. два входных полюса должны быть соединены с одним выходным.

Любая подстановка  $\lambda$  из  $N$  элементов может быть разбита на  $t$  подстановок  $\lambda_i$  ( $i=1, \dots, t$ ) из  $n_1, \dots, n_i, \dots, n_t$  элементов каждая так, что  $\bigcup_i n_i = N$  и  $\bigcap_i n_i = \emptyset$ .

Реализация в СМ подстановки  $\lambda$  путем последовательной реализации подстановок  $\lambda_1, \dots, \lambda_t$  называется реализацией за  $t$  частичных проходов.

В том случае, когда какой-либо из элементов структуры неисправен, процесс обработки информации может быть организован так, чтобы исключить (обойти) неисправный элемент.

Длиной обхода называется максимальный из диаметров графов, полученных из исходного удалением любого количества ребер, еще не нарушающим реберной связности графа.

Любая структура может быть размещена в единичной дискретной решетке, причем в одном узле решетки не более одного полюса элемента базиса (входного или выходного), связи между элементами только по образующим решетки и вдоль любой образующей может быть проведена только одна связь.

В этих предположениях площадью структуры называется минимальная площадь многоугольника, содержащего все компоненты структуры (все полюса элементов базиса и все связи между ними).

### Оценка основных параметров процессорных структур

В табл. 5.2 приведены оценки параметров ПС.

С точки зрения стоимости процессорной структуры (а это наиболее часто употребляемый показатель качества) оптимальными по порядку величины являются  $C_2(h, s)$  — кубосвязный цикл, все три класса деревьев  $T_0(n)$ ,  $T_1(n, k)$ ,  $T_2(n)$ , а также  $P(m, n)$ -структуры. Если учесть еще и величину отказоустойчивости, то здесь несомненным лидером оказываются  $P(m, n)$ -структуры, отказо-

Среднее расстояние $\bar{d} = \sum_{r=1}^D r N_r / N - 1$	Средняя плотность сообщения $\rho = \bar{d} N / Z$	Общее число связей $Z$	Отказоустойчивость $F$	Площадь $A$
$\asymp \sqrt{N}$	$\asymp \sqrt{N}$	$N$	3	$N$
$\asymp \log_2 N$	$\log_2 N$	$2N$	1	$O\left(\left(\frac{N}{\log N}\right)^2\right)$

устойчивость которых не ниже  $m-1$ . Однако эта структура (так же, как  $C_2(h, s)$ ) чрезвычайно неэффективно размещается в прямоугольных решетках, а значит, реализация ее в интегральном исполнении может вызвать затруднение при больших значениях  $N$ . По-видимому, структуры типа гипердерева и мультидерева, допускающие достаточно простую укладку и хорошо зарекомендовавшие себя как при решении комбинаторных задач типа сортировки, так и при реализации реляционных баз данных и задач типа лингвистического анализа [Goodman 1981; Arden et al., 1982], являются одними из первых кандидатов на реализацию в виде СБИС-структур. Но при этом следует иметь в виду, что окончательные выводы о преимуществах того или иного класса структур могут быть сделаны лишь на основании результатов моделирования пропускной способности структуры, ее отказоустойчивости, возникновения конфликтов, а также тщательного анализа и расчета возможностей реализации в конкретной интегральной технологии.

#### Функциональные возможности различных классов сетей межсоединений

**Топологическая эквивалентность сетей межсоединений.** Для СМ, имеющей  $N$  входов и  $N$  выходов ( $N=2^n$ ) и  $n$  рангов, выполняется свойство близости (buddy), если каждая пара  $\pi$ -элементов в  $i$ -м ранге связана только с одной парой  $\pi$ -элементов в  $(i+1)$ -м ранге [Dias et al., 1981].

Все неизбыточное, полнодоступные СМ, для которых выполняется свойство близости, топологически эквивалентны, если в каждом ранге такой сети размещено  $N/2$   $\pi$ -элементов с двумя входами и двумя выходами каждый [Wu et al., 1980; Agrawal, 1983].

Две СМ топологически эквивалентны, если изоморфны соответствующие этим сетям графы.

**Анализ и синтез сетей межсоединений.** Количество различных подстановок из  $N=2^n$  элементов равно  $N!$  Сеть ВСМ, при надлежащем выбранном состоянии, может реализовать любую из  $N!$  подстановок.

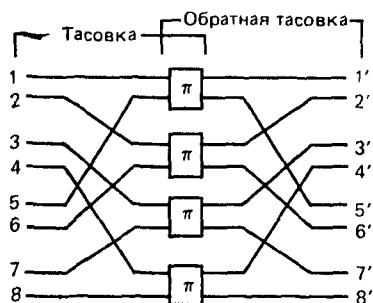
Пусть сети типа  $SCM(t)$ , состоящей из  $n$  или  $n-1$  рангов, поставлена в соответствие обратная сеть  $SCM^0(t)$ , состоящая из  $n-1$  или  $n$  рангов соответственно, такая, что подстановка  $\lambda_i$ , задающая соединения выходных полюсов  $\pi$ -элементов  $i$ -го ранга со входными полюсами  $\pi$ -элементами  $(i+1)$ -го ранга, совпадает с подстановкой, задающей соединения  $\pi$ -элементов  $(n-i)$ -го ранга с  $\pi$ -элементами  $(n-(i+1))$ -го ранга в исходной (прямой) сети  $SCM(t)$ .

Комбинация прямой  $SCM(t)$  и обратной  $SCM^0(t)$  сетей, имеющая  $2n-1$  рангов, может реализовать любую из  $(2^n)!$  подстановок [Agrawal, 1983].

Перестановка элементов входного алфавита, осуществляемая СМ, находящейся в любом состоянии, называется проходом.

Количество рангов в СМ можно «разменять» на число проходов. Известно, что:

Рис. 5.44. Одноранговая сеть типа тасовка-обмен — обратная тасовка



два прохода (один в прямом, а другой в обратном направлении) в любой сети SCM ( $t$ ) из  $n$  рангов позволяют реализовать любую подстановку без конфликтов [Agrawal, 1983];

в одноранговой сети типа тасовка-обмен — обратная тасовка (рис. 5.44) для реализации любой перестановки необходимо  $2n-1$  проход и достаточно  $3n-3$  проходов [Huang, Fripathi, 1986].

В SCM (1), состоящей из  $n$  рангов, при любом состоянии сети без конфликтов реализуется множество из  $2^{\lceil n/2 \rceil}$  подстановок [Agrawal, 1985].

Любую из  $(2^n)!$  подстановок можно реализовать в SCM ( $t$ ), используя не более  $2^{1/n/2}$  частных проходов [Agrawal, 1983].

Задача синтеза СМ состоит в том, чтобы для любой заданной перестановки входов синтезировать какую-либо СМ, реализующую эту перестановку без конфликтов. Для SCM (1) задача синтеза может быть решена с помощью дихотомии заданной перестановки [Agrawal, 1983].

Алгоритм решения задачи синтеза SCM (1). Обозначим символы входного алфавита  $1, 2, \dots, N$  и выходного —  $1', 2', 3', \dots, N'$ . Пусть требуемая перестановка  $\lambda$  задана в виде

$$\lambda: \begin{matrix} k_1 & k_2 & k_3 & \dots & k_N \\ 1' & 2' & 3' & \dots & N' \end{matrix}$$

(подстановки, заданные иначе, переупорядочением всегда можно привести к требуемому виду).

Существуют  $n$  последовательных дихотомий подстановки  $\lambda$  так, как показано на рис. 5.45. Требуемая сеть будет состоять из  $n$  рангов, в каждом из которых размещено  $N/2$   $\pi$ -элементов 1-го типа. Соединение между элементами всех рангов задаются в соответствии с дихотомией, показанной на рисунке. В первом ранге на входы первого элемента подаются переменные  $k_1 k_{N/2+1}$ , на входы второго —  $k_2 k_{N/2+2}$  и т. д. Во втором ранге на входы первого элемента подаются выходы первого элемента первого ранга и  $(N/4+1)$ -го элемента первого ранга и т. д.

На рис. 5.46 показан пример синтеза сети для случая  $N=8$  и подстановки

$$\begin{pmatrix} 2, & 6, & 4, & 7, & 1, & 5, & 8, & 3 \\ 1' & 2' & 3' & 4' & 5' & 6' & 7' & 8' \end{pmatrix}.$$

### Способы навигации и трансляции в процессорных сетях и сетях межсоединений

При организации передачи сообщений в вычислительных структурах определяющими являются два обстоятельства: способ управления элементами структуры и вычислительные возможности этих элементов.

Способ управления (централизованный или распределенный) указывает, как будут устанавливаться маршруты прохождения сообщений в сети: с помощью сигналов управления, вырабатываемых внешним по отношению к сети устройством управления, или с помощью алгоритмов управления, реализованных в каждом из элементов.

В первом случае элемент сети не тратит свои вычислительные ресурсы на алгоритмы управления, но необходимо иметь достаточно сложное централизованное устройство управления, надежность которого и будет определять фактически надежность всей структуры.

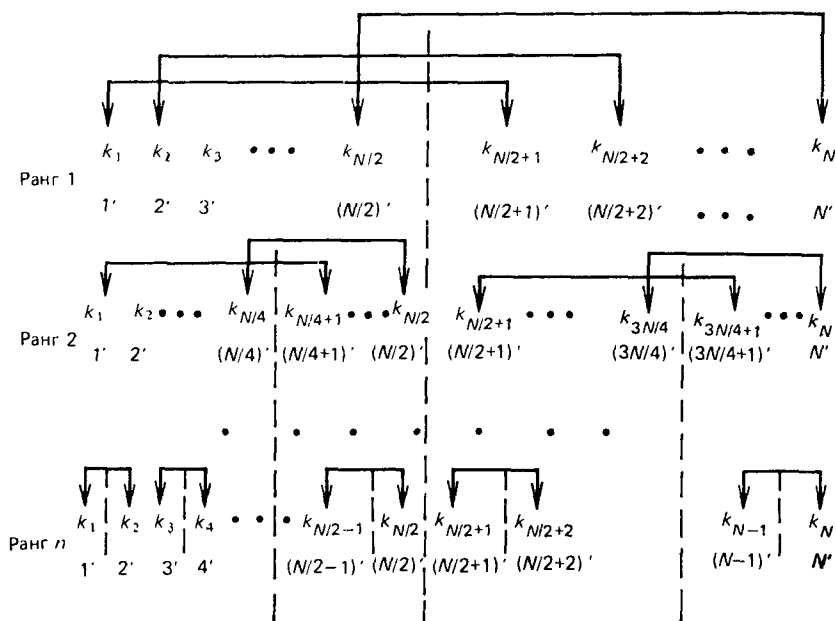


Рис. 5.45. Сеть последовательных дихотомий подстановки

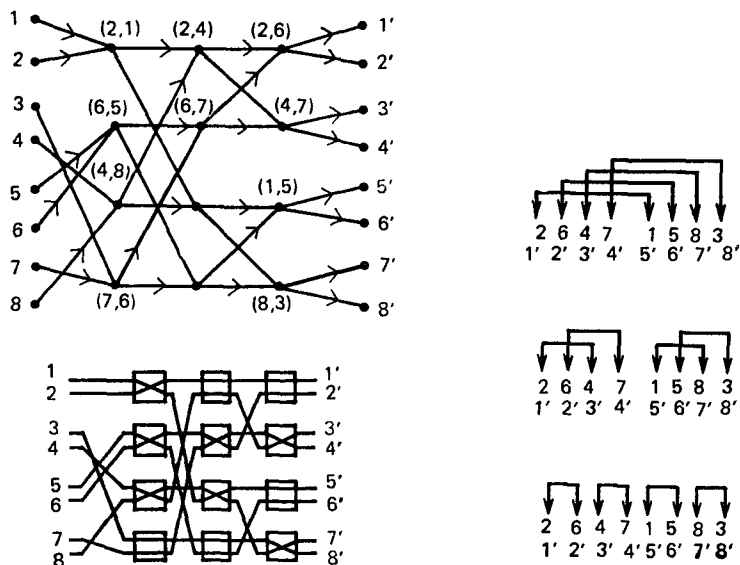


Рис. 5.46. Синтез сети



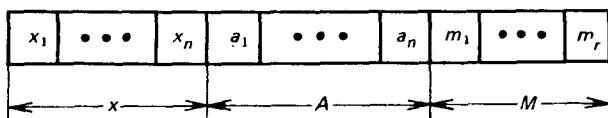


Рис. 5.47. Формат сообщения для сети  $C_0(n)$

Во втором случае от элемента сети могут потребоваться довольно значительные ресурсы производительности и памяти на выполнение алгоритмов навигации, но сеть обладает свойством «мягкой деградации» — выход из строя каких-то элементов сети не приводит к катастрофическому отказу.

**Навигация и трансляция в процессорных сетях.** В процессорных сетях, как правило, предполагается распределенный способ управления. Рассмотрим способ навигации для сети  $C_0(n)$ .

Структура передаваемого сообщения показана на рис. 5.47. Каждый элемент  $C_0(n)$  хранит свой двоичный номер  $X$  и содержит устройство, позволяющее сравнивать адрес  $A$  места назначения пришедшего сообщения с номером  $X$ .

**Шаг 1.** Сравнение адреса  $A$  и номера  $X$ . При совпадении конец алгоритма. Иначе — шаг 2.

**Шаг 2.** Фиксируется первая несовпадающая координата, т. е. первый бит, для которого  $x_i \neq a_i$ , и сообщение отправляется по связи, соответствующей несовпадающей координате. Возвращение к шагу 1.

При таком алгоритме сообщение не позднее чем через  $n$  шагов достигает адресата. Поскольку в  $C_0(n)$  любая вершина соединена с любой другой  $n$  дизъюнктивными путями одинаковой длины, даже при наличии  $(n-1)$ -й неисправной связи сообщение достигнет адресата по кратчайшему пути.

В нерегулярной сети  $C_1(m_1, \dots, m_n)$  алгоритм навигации совпадает с алгоритмом для сети  $C_0(n)$  с той разницей, что номер несовпадающей цифры адреса  $a_i \neq x_i$  определяет координату, вдоль которой отправляется сообщение, а абсолютное значение  $|a_i - x_i|$  определяет адрес вершины вдоль этого направления [Pradhan, 1985].

Так же, как и для  $C_0(n)$ , в структуре  $C_1(m_1, \dots, m_n)$  существует  $n$  дизъюнктивных путей одинаковой и кратчайшей длины  $n$  между любой парой вершин  $x_i, a_j$ . Кроме того, между любой парой вершин существует  $l$  альтернативных путей, где  $l = \sum_{i=1}^n (m_i - 1)$  — степень вершины. Таким образом, при любом

числе неисправных связей меньше чем  $l$  между любой парой вершин можно передать сообщение и длина маршрута будет не больше  $n+1$ .

Для сети  $P(m, n)$  формат сообщений показан на рис. 5.48. При послыке сообщения в поле теговых бит  $T$  заносится адрес места назначения. Алгоритм навигации выглядит следующим образом.

**Шаг 1.** Сравнение адреса  $X$  текущей вершины с адресом места назначения  $A$ . При совпадении конец алгоритма. Иначе шаг 2.

**Шаг 2.** Сравнение  $x_0$  (последнего значащего разряда  $X$ ) с первым битом адреса назначения  $A$ , записанного в поле  $T$ . При равенстве — шаг 3, иначе сообщение пересылается к вершине, адрес которой  $(x_{n-1}, \dots, x_1, \bar{x}_0)$ .

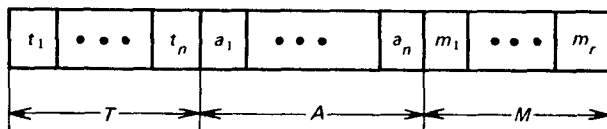


Рис. 5.48. Формат сообщения для сети  $P(m, n)$

Шаг 3. Теговое поле  $T$  сдвигается влево на один бит. Если номер вершины  $x$ , в которой находится сообщение, равен 0, или  $N-1$ , возврат к шагу 2. Иначе, сообщение пересылается к вершине, адрес которой  $(x_{n-2}, \dots, x_1, x_0, x_{n-1})$ .

В результате описанного алгоритма сообщение совершает следующий путь от источника  $S=(S_{n-1}, \dots, S_1, S_0)$  к месту назначения  $A=(a_{n-1}, \dots, a_1, a_0)$ :

Способы трансляции для сетей  $C_0(n)$  и  $C_1(m_1, \dots, m_n)$ . Любой элемент в этих сетях может осуществить трансляцию в точности за  $n$  шагов следующим образом [Bhuayan, Agrawal, 1984].

Обозначим через  $i$  номер координаты ( $1 \leq i \leq n$ ) и будем считать без потери общности, что адрес элемента источника сообщений  $S = (0, 0, \dots, 0)$ . Этот элемент посылает сообщение вдоль каждой из координат  $i$ , причем сообщения, poslanному вдоль координаты  $i$ , присписан вес  $i$ . Вершины, расположенные вдоль  $i$ -й координаты (в случае  $S_0(n)$  — одна вершина), получив сообщение, уменьшают его вес на единицу и посылают вдоль всех тех координат, чьи номера не меньше  $i-1$ . При этом каждому посланному сообщению присписывается вес, равный номеру координаты, вдоль которой он посылается.

**Навигация в сетях межсоединений.** В сетях межсоединений, состоящих из элементов 1-го и 2-го типа, из-за простоты таких элементов применяется централизованное управление. При этом устройство управления хранит (или вычисляет) набор управляющих сигналов, необходимый для того, чтобы СМ реализовала требуемую перестановку входов или передачу сообщения от входа  $x$  к выходу  $y$ .

При распределенном управлении решение о том, куда передать входное сообщение, принимает сам  $\pi$ -элемент. Это решение может быть принято на основании некоторой табличной информации, содержащейся в памяти  $\pi$ -элемента, с помощью некоторого алгоритма вычисления или случайным образом.

Входное сообщение должно нести информацию об адресе назначения и некоторую дополнительную информацию, зависящую от типа  $\pi$ -элемента.

Рассмотрим сеть SCM (2), для которой формат сообщения имеет вид, показанный на рис. 5.49.  $\pi$ -элемент в ранге  $i$ , получивший такое сообщение, анализирует  $i$ -й бит как в маске трансляции, так и в адресе назначения. Если бит трансляции равен 1, то сообщение передается на оба выхода. Если бит трансляции равен 0, то значение соответствующего бита адреса назначения определяется, на верхний или нижний выход  $\pi$ -элемента необходимо передать сообще-

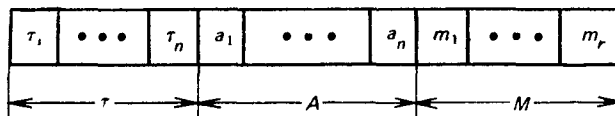


Рис. 5.49. Формат сообщения для сети SCM(2)

ние. Управление передачей сообщений осуществляется с помощью асинхронных протоколов. Каждый  $\pi$ -элемент также проводит проверку правильности полученных сообщений, которые кодируются в некотором избыточном коде (контроль по четности, контрольные суммы) [Raghavendia, 1986].

### Тестовое диагностирование и обеспечение отказоустойчивости

Наиболее полные результаты имеются относительно тестового диагностирования сетей межсоединений и процессорных структур типа  $S_2$ . Приведем оценки длины обнаруживающих тестов и тестов поиска для константных неисправностей и сетей SCM (1) [Feng, Wu, 1981].

Для неисправностей из класса одиночных констант и SCM (1) точная оценка длины обнаруживающего теста равна 4. Если неисправность проявляется только на одном выходном полюсе SCM (1), то длина теста поиска не превышает  $6 + 2 \lceil \log_2 \log_2 N \rceil$ .

Для неисправностей из класса кратных констант и SCM (1) длина обнаруживающего теста не более  $2(1 + \lceil \log_2 N \rceil)$ .

Для неисправностей из класса одиночных констант и CM типа SCM (2) длина обнаруживающего теста не более чем  $\max(28; 4 \log_2 N + 8; 2 \lceil \log_2 N \log_2 N \rceil)$ , причем тест такой длины в большинстве случаев позволяет определить номер неисправного  $\pi$ -элемента. Тестовый эксперимент в этом случае состоит из трех фаз: первые две фазы совпадают с экспериментом для SCM (1) — два входных набора в «прямом» состоянии всех  $\pi$ -элементов и два в состоянии «обмена», а в третьей фазе проверяются состояния «трансляции»  $\pi$ -элементов, т. е. состояния  $S_3$  и  $S_{12}$ . Для этого элементы каждого ранга CM независимо от остальных устанавливаются в состояние  $S_3(S_{12})$ , в то время как элементы остальных рангов остаются в «прямом» состоянии или в состоянии «обмена».

**Отказоустойчивость в сети межсоединений.** Сеть межсоединений будет отказоустойчивой (по отношению к множеству неисправностей  $F$ ), если при возникновении в сети любой неисправности из  $F$  существует возможность реализовать требуемую перестановку входных переменных. Обеспечить отказоустойчивость можно за счет схемной или временной избыточности.

Возможность обеспечения отказоустойчивости по отношению к множеству  $F$  существенно зависит от типа CM. Например, в полностью доступной CM, где существует только один путь от любого входного полюса к любому выходному, одиночная константная неисправность в ранге  $n/2$  может привести к недоступности  $1/4$  всех выходных полюсов. Если в CM существуют альтернативные пути от любого входного полюса к любому выходному (сети ACM или BCM), определенные классы входных перестановок могут быть реализованы и в случае константных неисправностей.

Для гарантированной коррекции неисправностей определенных классов необходимо (даже для CM с альтернативными путями) введение дополнительных  $\pi$ -элементов.

Одиночная константная неисправность управляющего входа в BCM может быть скорректирована с помощью одного дополнительного  $\pi$ -элемента на входе или выходе CM [Agrawal, 1982]. Это объясняется тем, что для BCM одиночная неисправность управляющего входа  $\pi$ -элемента приводит к невозможности реализации не более чем двух входных перестановок (из-за неисправности только два выхода меняются местами). Поэтому добавление одного  $\pi$ -элемента, на входы которого подаются «перепутанные» выходы, достаточно для получения правильной перестановки.

Для сетей типа SCM (2) с распределенным управлением отказоустойчивость, при которой  $F$  состоит из одиночных константных неисправностей, можно обеспечить добавлением еще одного  $(n+1)$ -го ранга из  $\pi$ -элементов и схем мульти-

<sup>1</sup> Здесь и далее  $\lfloor x \rfloor$  — ближайшее к  $x$  целое число с недостатком, а  $\lceil x \rceil$  — ближайшее к  $x$  целое с избытком.

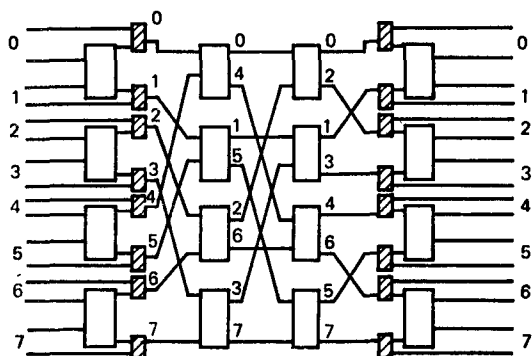


Рис. 5.50. Схема из  $\pi$ -элементов

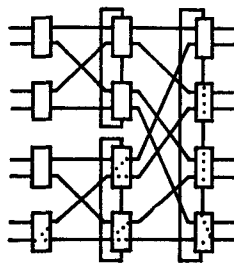


Рис. 5.51. Схема из модифицированных  $\pi$ -элементов

плексирования/демультиплексирования на входе и выходе СМ (заштрихованные прямоугольники на рис. 5.50). Кроме того, все входные и выходные связи удваиваются (на рис. 5.50 показан пример для  $N=8$ ). В этом случае ранг с неисправным  $\pi$ -элементом можно исключить из конфигурации СМ [Adams, Agrawal, Seigel, 1987].

Более общий способ введения схемой избыточности связи со специальным усложнением каждого  $\pi$ -элемента СМ и введением альтернативных путей передачи данных для обхода неисправных элементов сети. Например, стандартный  $\pi$ -элемент типа 1, 2 или 3 может быть модифицирован введением третьего входа и выхода, при этом общее число состояний такого элемента равно  $2^{3 \cdot 3} = 512$ . Сеть типа SCM (1) на восемь входов и выходов из таких модифицированных  $\pi$ -элементов показана на рис. 5.51.

Элементы ранга  $i$  ( $0 \leq i \leq n-1$ ) разбиты на  $2^i$  непересекающихся подмножеств, в каждом из которых  $2^{n-i-1}$  элементов. Все элементы, принадлежащие к одному подмножеству, соединены между собой так, как показано на рис. 5.51. Так как входы  $x_1$  и  $x_2$  могут быть соединены не только с  $y_1$  и  $y_2$ , но и с дополнительным выходом  $u$ , появляются возможности создания альтернативных путей от любого входа к любому выходу. Наиболее эффективно применение такой схемной модификации при распределенном управлении в СМ и, следовательно, при достаточно сложных  $\pi$ -элементах. В этом случае, анализируя входное сообщение,  $\pi$ -элемент может принять решение о передаче этого сообщения на верхний или нижний выход, при наличии сигнала о конфликте или неисправности на дополнительный выход  $u$ .

Введение временной избыточности для достижения отказоустойчивости означает возможность обхода неисправных  $\pi$ -элементов в СМ при организации нескольких проходов от входных к выходным полюсам СМ и обратно.

Пусть СМ предназначена не для реализации фиксированного множества подстановок, а для передачи сообщений от  $i$ -го входного полюса к  $j$ -му выходному. Причем эта передача может быть осуществлена не только непосредственно, но и через другие выходные полюсы, т. е. организацией маршрутов  $i_1 j_2 \dots j$ . Сеть межсоединений обладает свойством динамической полнодоступности, если для всех вход-выходных пар  $(i, j)$  существуют цепочки  $(i_1 \dots j_k j)$  конечной длины. Хотя единственная неисправность в СМ типа SCM ( $t$ ) нарушает свойство полнодоступности, свойство динамической полнодоступности может сохраняться при достаточно большом числе неисправных элементов.

Пусть неисправность  $\pi$ -элемента в СМ означает, что оба его выхода оказываются недоступными. Множество таких неисправных элементов называется некритическим, если оно не нарушает динамической полнодоступности.

Для СМ типа SCM (2), имеющей  $N=2^n$  входных и выходных полюсов, множество неисправных элементов  $F$  является некритическим, если выполнены условия:

все элементы первого и последнего ранга исправны;  
 $|F_{\max} \cap A_{ij}| \leq 2^{n-i-2}$  для всех рангов  $i$  ( $1 \leq i \leq n-2$ ) и для всех элементов  $j$  в ранге  $i$ , где  $0 \leq j \leq 2^i - 1$ . Здесь множество элементов

$$\{A_{ij}\} = \{(i, j), (i, j + 2^i), (i, j + 2^{i+1}), \dots, (i, j + \frac{N}{2} - 2^i)\}$$

для  $1 \leq i \leq n-2$  и  $0 \leq j \leq 2^i - 1$ , т. е. множество  $\{A_{ij}\}$  в ранге  $i$  разбивает все  $\pi$ -элементы на  $2^i$  непересекающихся подмножества мощностью  $2^{n-i-1}$  каждое. Иными словами, пересечение множества неисправных элементов с множествами  $A_{ij}$  должно содержать не более половины элементов  $A_{ij}$ .

Из приведенных выше условий видно, что максимальное число неисправных  $\pi$ -элементов, не нарушающих динамической полидоступности, равно  $N(\log_2 N - 1)/4$ . В любой СМ типа SCM (2), в которой множество неисправных элементов не является критическим, сообщения могут быть переданы от любого входа к любому выходу не более чем за  $\lceil \log N/2 \rceil$  проходов для  $N \leq 16$  и не более чем за  $(\log N - 2)$  прохода для  $N \geq 32$ .

**Отказоустойчивость в процессорных сетях.** В процессорных сетях отказоустойчивость может быть достигнута введением в сеть избыточных  $\Pi$ -элементов и организацией обхода неисправного элемента (исключения его из конфигурации) без потери функциональных возможностей в сети.

В зависимости от организации сети возможно два принципиально различных подхода к процедурам перестройки (реконфигурации) сети. Эти подходы по аналогии с принципами управления можно назвать централизованным и распределенным.

В первом случае номера неисправных элементов предполагаются априорно известными. Для получения такой информации процессорная сеть, как правило, должна допускать возможность произвольного доступа к каждому элементу со стороны внешних входов (адресацию каждого элемента). Во втором случае информацию о неисправности элемента сети можно получить только в процессе навигации, пытаясь передать сообщение от одного элемента сети к другому.

Пусть для сети  $P(m, n)$  известны номера неисправных элементов. Тогда относительно длины обхода в этой сети известно следующее [Pradhan, 1985].

**Случай  $P(2n)$ .** Существует алгоритм навигации  $A_1$ , такой, что при наличии любой одиночной неисправности длина маршрута  $l(u, v)$  для любой пары вершин  $u, v$  не превышает  $4n-1$ . Существует алгоритм навигации  $A_2$ , такой, что при наличии любых двух неисправностей длина маршрута  $l(u, v)$  для любой пары вершин  $u, v$  не превышает  $4n+2$ .

**Случай  $P(m, n)$  ( $m \geq 3, n \geq 3$ ).** Существует алгоритм навигации  $A_3$ , такой, что при наличии  $t \leq (m-1)/2$  неисправностей длина маршрута не превышает  $2n-1+11t$ .

Существует алгоритм навигации  $A_4$ , такой, что при наличии больше чем  $(m-1)/2$  неисправностей (но меньше  $m-1$ ) длина маршрута не превышает  $6n-3$ .

### Классификация процессорных перестраиваемых структур; примеры построения

Перестраиваемые процессорные структуры (ППС) могут иметь в своей основе любую процессорную структуру. Выбор топологии такой базовой структуры зависит от способа предполагаемой физической реализации ППС и характера решаемых задач. Так, если предполагается реализовать ППС в виде специализированной СБИС, предпочтение отдается регулярным, плотно упакованным структурам типа деревьев или решеток.

После того как выбрана базовая процессорная структура, необходимо определить такие существенные особенности ППС, как способы введения избы-

точности, способы диагностирования, возможные алгоритмы реконфигурации. Каждую ППС  $W$  опишем пятеркой вида  $\{D, O, P, Z, B\}$ , где

$D$  характеризует способ диагностирования и принимает значения  $\{1, 2\}$ , причем 1 означает, что в ППС возможно лишь внешнее тестовое диагностирование, а 2 означает возможность автономного тестового диагностирования самотестирования;

$O$  характеризует тип отказоустойчивого поведения и принимает значения  $\{1, 2\}$ , причем 1 означает, что структура считается исправной только при некотором фиксированном числе  $P$ -элементов, а 2 означает наличие более чем одного «состояния» исправности по числу  $P$ -элементов, т. е. допускается «мягкая деградация».

$P$  характеризует способ реконфигурации и принимает значения  $\{1, 2, 3, 4\}$ , причем 1 означает, что в ППС содержатся  $\pi$ -элементы, независимые от  $P$ -элементов, и с их помощью осуществляется обход тех  $P$ -элементов, которые следует исключить из структуры; 2 означает, что структура содержит более простые, локальные  $\pi$ -элементы, а  $P$ -элемент в любом состоянии допускает передачу информации в любом разрешенном направлении; 3 означает, что в ППС реализована шинная архитектура и исключение из структуры происходит при отключении от общей шины; 4 означает, что в ППС реализован адресный произвольный доступ к любому  $P$ -элементу.

$Z$  характеризует способ замещения неисправных элементов структуры и принимает значения  $\{1, 2, 3\}$ , причем 1 означает наличие прямого, локального способа замещения, т. е. такого, когда неисправный  $P$ -элемент замещается ближайшим исправным; 2 означает наличие глобального способа замещения, когда исключение неисправного  $P$ -элемента приводит к реконфигурации всей ППС; 3 означает, что неисправный элемент удаляется из ППС без замещения.

$B$  характеризует вид алгоритма реконфигурации и принимает значения  $\{1, 2\}$ , причем 1 означает применение алгоритма статической реконфигурации, проводимой после изготовления и имеющей, как правило, необратимый характер (например, лазерное пережигание связей); 2 означает применение алгоритма динамической реконфигурации, т. е. реконфигурации, которую можно осуществлять неоднократно в процессе эксплуатации.

Важной количественной характеристикой ППС  $W$  является сложность  $\Omega(W)$  реконфигурации

$$\Omega(W) = \lceil \log_2 S(W) \rceil,$$

где  $S(W)$  — суммарное число состояний всех  $\pi$ -элементов ППС.

Нижняя оценка  $\Omega(W)$  определяется количеством различных правильных конфигураций, которые можно получить в ППС  $W$ , т. е. таких, при которых ППС считается исправной.

Если  $R$  — число правильных конфигураций ППС  $W$ , то  $S(W) \geq R(W)$ . Значение  $S(W)$  зависит от числа избыточных элементов, способа реконфигурации и способа определения понятия «правильная» конфигурация.

**Пример 5.1.** Пусть базовой структурой для ППС  $W$  является структура двумерной решетки. В узлах решетки размещены  $\pi$ - или  $P$ -элементы, причем в исходном состоянии  $W$  — это прямоугольная матрица, содержащая  $M$  строк по  $L$   $P$ -элементов в каждой. С помощью  $\pi$ -элементов необходимо в такой матрице  $M \times L$  выделять различные конфигурации, которые можно считать правильными.

Пусть правильной будет любая конфигурация  $W_1$ , которая содержит ровно  $N^2$   $P$ -элементов.

Обозначим  $M = N + a$ ;  $L = N + b$ ;  $a \geq 0$ ,  $b \geq 0$ ;  $a + b \geq 1$  и  $a, b \leq N$ . Тогда  $R(W_1) = C_{(N+a)(N+b)}^{N^2}$ . Так как  $S(W_1) \geq R(W_1)$ , то

$$\log_2 S(W_1) \geq 2(a+b)N \log_2 N.$$

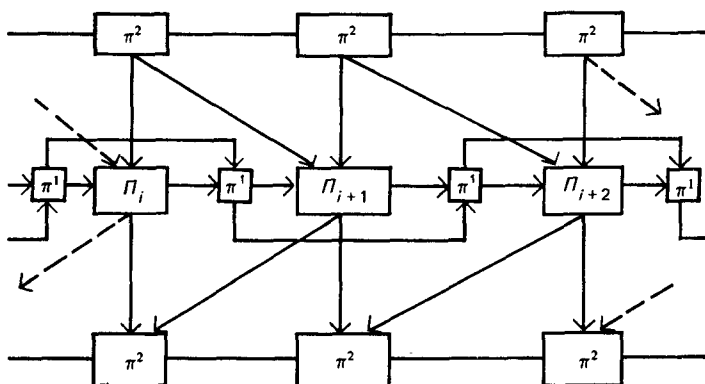


Рис. 5.52. Фрагмент ППС

Пусть в исходной матрице содержится только  $a$  избыточных столбцов, т. е. исходная матрица имеет размерность  $(N+a) \times N$ . Назовем правильной любую конфигурацию  $W_2$ , при которой в каждой строке содержится  $N$   $\Pi$ -элементов.

Тогда  $R(W_2) = (C_{(N+a)N}^N$

$$\log_2 S(W_2) \geq aN \log_2 (N+a).$$

Пусть правильной является такая конфигурация, которая представляет собой квадратную матрицу, содержащую  $N$  строк и  $N$  столбцов. Иными словами, правильной является только такая конфигурация, которая получается удалением  $a$  столбцов и  $b$  строк в исходной матрице  $(N+a) \times (N+b)$ . Тогда

$$R(W_3) = (C_{N+a}^a) (C_N^b b);$$

$$\log_2 S(W_3) \geq a \log_2 (N+a) + b \log_2 (N+b).$$

Как видно из рассмотренного примера, суммарная сложность  $\pi$ -элементов существенно зависит от того, какие конфигурации принято считать допустимыми. Достигается ли нижняя граница сложности реконфигурации в конкретной ППС, зависит уже от того, какие конкретно выбраны  $\pi$ -элементы и как реализован алгоритм реконфигурации.

**Пример 5.2.** Построим модель ППС, относящейся к типу  $\{1, 1, 1, 1, 2\}$ , предположив, что задана исходная матрица процессорных элементов  $(N+1) \times N$  и необходимо реконфигурировать такую матрицу в матрицу  $N \times N$ , исключая любой (один) элемент из каждой строки.

На рис. 5.52 показан фрагмент ППС, где в каждой строке чередуются  $\Pi$ -и  $\pi^1$ -элементы, которые могут находиться в состояниях  $S_1$  и  $S_2$  (см. рис. 5.26). Наличие таких  $\pi$ -элементов позволяет обойти любой из  $\Pi$ -элементов строки. Над и под каждой строкой размещены  $\pi^2$ -элементы, каждый из которых может находиться в четырех состояниях  $S_1, S_2, S_4$  и  $S_8$ . Эти  $\pi$ -элементы позволяют обойти любой из процессоров по вертикали.

Внешняя система диагностики в такой ППС и внешняя система управления всеми  $\pi$ -элементами позволяют определить номера тех  $\Pi$ -элементов в каждой строке, которые должны быть исключены из конфигурации. Затем система управления задает на каждый  $\pi$ -элемент структуры соответствующую команду для того, чтобы осуществить реконфигурацию.

**Пример 5.3.** Построим модель ППС, относящейся к типу  $\{1, 1, 2, 1, 2\}$ , т. е. отличающуюся от предыдущей тем, что  $\Pi$ -элементы в любом состоянии могут правильно передать информацию с вертикального входа на вертикальный выход. В этом случае исходная матрица процессорных элементов имеет  $N+1$

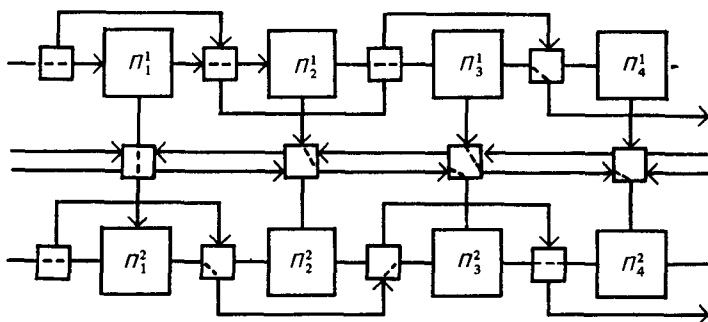


Рис. 5.53. Фрагмент ППС с  $\Pi$ -элементами на 3 входа и 3 выхода

строку и  $N+1$  столбец. Исходную матрицу  $(N+1) \times (N+1)$  необходимо реконфигурировать в матрицу  $N \times N$ , исключив любую (одну) строку и любой (один)  $\Pi$ -элемент в каждой из  $N$  оставшихся строк.

На рис. 5.53 показан фрагмент ППС, где в каждой строке размещены такие же  $\pi$ -элементы, как и в предыдущем примере, а между строками размещены  $\pi$ -элементы с тремя входами и тремя выходами, каждый из которых может находиться в семи состояниях, показанных на рис. 5.54. Такие  $\pi$ -элементы позволяют осуществить любую требуемую реконфигурацию. На рис. 5.53 из конфигурации исключен четвертый элемент первой строки и второй элемент второй строки.

**Пример 5.4.** Построим модель ППС, относящейся к типу  $\{1, 1, 1, 2, 2\}$ , т. е. с возможностью глобального способа замещения неисправных элементов. Исходная матрица процессорных элементов, как и в предыдущем примере, имеет вид  $(N+1) \times (N+1)$ . Эту исходную матрицу необходимо реконфигурировать в матрицу  $N \times N$ , исключив любую (одну) строку и любой (один) столбец. Таким образом, в подобной ППС можно исключить не более двух произвольно расположенных  $\Pi$ -элементов, в то время как в ППС примера 5.3 — три произвольно расположенных  $\Pi$ -элемента.

На рис. 5.55 показан фрагмент (один столбец) такой ППС. Реконфигурация осуществляется за счет  $\pi$ -элементов, расположенных на периферии структуры и являющихся схемами мультиплексирования (демультиплексирования) на  $N+1$  входов (выходов) каждая. Эти периферийные схемы превращают каждый столбец и строку в тор. Если необходимо из такого тора исключить, например, второй  $\Pi$ -элемент, осуществляется перенумерация всех элементов (третий элемент становится первым, четвертый — вторым, а первый — третьим).

Создание сложных ПС в виде специализированных СБИС с числом активных элементов  $10^7$  и более неизбежно приводит к необходимости организации этой структуры в виде избыточной перестраиваемой структуры. Это единственный способ обеспечить приемлемый показатель «выход годных» кристаллов, т. е. долю исправных кристаллов среди всех изготовленных. Изучение зависимости выхода годных от вероятности неисправности элементов ППС при различных типах перестраиваемых структур возможно лишь с помощью численного моделирования достаточного большого объема. В качестве иллюстрации полученных результатов на рис. 5.56 показано поведение доли выхода годных  $Y$  в зависимости от вероятности исправности  $p$  одного  $\Pi$ -элемента для безызыточной структуры  $10 \times 10$  ПЭ (сплошная линия) и ППС типа рассмотренной в примере 5.2 и содержащей  $13 \times 10$  ПЭ (штриховая линия). При расчетах надежности и моделировании  $\pi$ -элементы предполагались неисправными с вероятностью  $p_\pi = 1 - \exp(-\beta \ln(1-p))$ , где  $\beta$  — отношение площади  $\pi$ -элемента к площади  $\Pi$ -элемента.



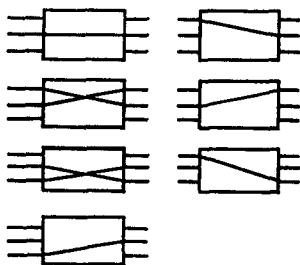


Рис. 5.54. Состояния П-элемента на 3 входа и 3 выхода

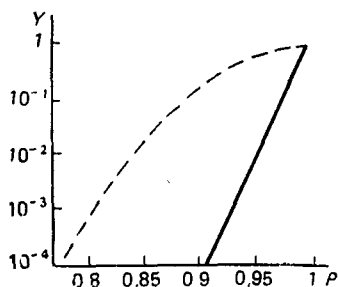


Рис. 5.56. Зависимости выхода годных от вероятности исправности одного П-элемента для двух структур

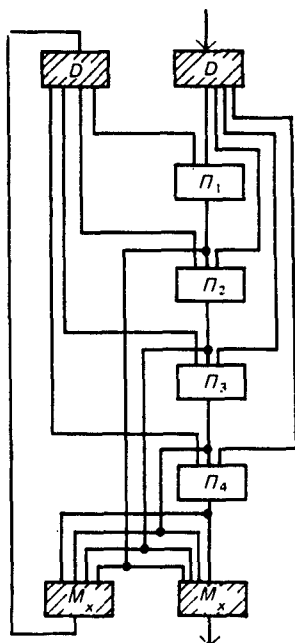


Рис. 5.55. Столбец ППС

Из рис. 5.56 видно, что даже для такой относительно небольшой решетки выход годных ( $Y$ ) не менее 10% достигается в безызыточной структуре при  $P > 0,98$ , а при  $P < 0,95$  становится практически неаблюдаемой величиной (сплошная линия). В то же время выход годных не менее 10% ППС  $13 \times 10$  можно обеспечить уже при  $P > 0,88$  (штриховая линия) [Горяшко и др., 1988].

### Перспективы и прогноз

Одна из наиболее устойчивых тенденций в технике, связанной с процессами переработки информации, состоит в уменьшении стоимости создания аппаратных средств при сохранении почти неизменной стоимости разработки программных средств. Поэтому для все более широкого круга задач, традиционно решавшихся программными средствами (например, создание трансляторов для языков высокого уровня), обсуждаются возможности разработки специализированных СБИС [Хартман, 1986].

Для большинства классических задач ИИ весьма перспективным представляется аппаратное воплощение процессов их решения в виде заказных СБИС. Объясняется это следующим:

уверенностью многих исследователей в том, что именно задачи распознавания, моделирования познавательной деятельности, вычисления в семантических сетях наилучшим образом распараллеливаются [Fennell, Lesser, 1977];

возможностью для многих задач, прежде всего связанных с поиском и сортировкой, предложить адекватные специализированные структуры. Хорошим

примером могут служить проекты словарных машин [Atallah, Kosaraju, 1985]; в структурах типа гипердерева удастся реализовать операции поиска с временными затратами  $O(\log n)$ , где  $n$  — текущее значение количества элементов данных, записанных в словаре емкостью  $N$  (емкость словаря — это количество пар <ключ, запись>);

все более распространяющимся скептицизмом в отношении возможностей программной реализации практически интересных задач ИИ даже на самых мощных, многопроцессорных универсальных вычислительных системах.

Оценивая перспективы аппаратной реализации сложных задач ИИ, необходимо иметь прогноз состояния дел в области интегральной технологии. На протяжении последних двадцати лет развитие интегральной технологии опережало самые оптимистические прогнозы. Так, на конференции Калифорнийского технологического института высказывались предположения о том, что к началу 90-х годов должны появиться кристаллы, содержащие до  $10^7$  активных элементов. Но уже в середине 1986 г. фирма TRW объявила [Электроника, 1986], что к концу 1988 г. будут изготовлены шесть типов кристаллов со следующими параметрами: четырехпортовое ЗУ с произвольной выборкой емкостью 64К 32-разрядных слов (число активных элементов  $27,9 \cdot 10^6$ , коэффициент избыточности 1,48, ожидаемый выход годных 50%); процессор БПФ (число активных элементов  $3,9 \cdot 10^6$ , коэффициент избыточности 1,77, ожидаемый выход годных 38%); сигнальный процессор (число активных элементов  $27,9 \cdot 10^6$ , коэффициент избыточности 2,84, ожидаемый выход годных 29%); процессор данных (число активных элементов  $34,7 \cdot 10^6$ , коэффициент избыточности 1,59, ожидаемый выход годных 20%); ассоциативный процессор (число активных элементов  $10,1 \times 10^6$ , коэффициент избыточности 1,42, ожидаемый выход годных 45%); цифровой коррелятор (число активных элементов  $10 \cdot 10^6$ , коэффициент избыточности 1,61, ожидаемый выход годных 42%).

Из этих кристаллов только четырехпортовое ОЗУ можно отнести к традиционным по архитектуре, если не учитывать почти 50%-ную избыточность. Остальные типы кристаллов — это различные перестраиваемые процессорные структуры, причем ассоциативный процессор, если судить по названию, ориентирован на задачи, необходимые для систем, работающих со знаниями, и экспертных систем.

Таким образом, уже в ближайшие годы можно ожидать, что создание специализированных СБИС с 30—40 млн. активных элементов окажется не только технически, но и экономически реальным делом. А это означает, что на одном кристалле будет возможно размещение структуры типа гипердерева, имеющего около  $2^6$  листьев (для словарных машин) или  $2^{16}$  процессорных модулей, соединенных с таким же количеством модулей памяти отказоустойчивой сетью межсоединений или, наконец, структуры решетки, содержащей несколько тысяч процессорных и переклюочательных элементов и образующих перестраиваемую структуру для решения задач распознавания.

Ориентируясь на подобные возможности интегральной технологии, специалисты в области машин ИИ уже создают проекты сотовых процессорных структур с различными механизмами общения процессоров в структуре для моделирования механизмов зрения и процессоров вычисления в семантических сетях, а также многопроцессорные устройства с шиной архитектурой для экспертных систем или структуры типа  $C_0$  с числом вершин  $2^{16}$  для задач, связанных с обработкой знаний [Computer, 1987]. Характерно, что подобные проекты или даже опытные образцы создаются не только в университетских центрах (Cambridge, Carnegie Mellon University, University of Southern California), но и в таких промышленных фирмах, как Fairchild, Palo Alto или Xerox, California.

Переход интегральной технологии в субмикронный диапазон в сочетании с развитием принципов создания отказоустойчивых архитектур и создание оптических ЗУ с лазерным считыванием очень большой емкости позволяет предположить, что к концу 90-х годов будут созданы аппараты реализованы компактные совершенные устройства машинного зрения, распознавания речи и, возможно, общения с ЭВМ на естественном языке.

Гораздо менее определенными выглядят перспективы, связанные с попытками аппаратно смоделировать поведение нейронных сетей. Несмотря на проявляющийся интерес к подобным занятиям [Hopfield et al., 1986], здесь ощущается, с одной стороны, отсутствие принципиально новых идей, а с другой — весьма большие технологические трудности. Если говорить об идейной стороне вопроса, то исследователи толпятся вокруг положений, выдвинутых еще в 60-е годы и разработанных советскими учеными. Эти идеи предполагают создание таких систем, для которых поиск решения задачи сводится к достижению состояния, соответствующего локальному энергетическому минимуму системы. Основная трудность интегральной реализации нейроноподобных структур в создании интегральных структур с большим числом внутренних связей. Эту трудность в какой-то мере можно будет преодолеть в случае технологического прорыва в области разработки трехмерных кремниевых структур.

## **5.5. Средства обработки нечеткой информации**

*Л. С. Берштейн, С. Я. Коровин, А. Н. Мелихов*

### **Основные положения**

По мере возрастания интереса к созданию интеллектуальных систем повышается интерес к использованию в них нечеткой логики. Известен ряд успешных приложений нечеткой логики для решения задач управления технологическими процессами [Tong, 1978; Кафаров и др., 1978]. До сих пор, однако, все системы, базирующиеся на нечеткой логике, предназначенные для принятия решений при управлении технологическими процессами, представляли собой программные комплексы, реализуемые на ЭВМ с традиционной архитектурой. В последнее время начаты разработки специализированных аппаратных средств выполнения нечетких логических функций, а также аппаратно-программных комплексов обработки нечеткой информации и реализации нечетких алгоритмов и моделей. После появления основополагающих работ по теории нечетких множеств [Zadeh, 1965; Кофман, 1982] начаты разработки дискретных вычислительных устройств для реализации функций нечеткой логики, которые развиваются в двух направлениях.

1. Разработка элементной базы нечетких вычислительных машин [Балашов и др., 1986; Борисов, 1986; Карелин и др., 1983, 1984]. К сожалению, разработка логических элементов нечетких вычислительных машин продвигается пока довольно медленно. Это связано, во-первых, с тем, что наряду с минимальным базисом нечеткой логики существует несколько других базисов и продолжается дискуссия о достоинствах и недостатках того или иного способа реализации операций нечеткой логики. Во-вторых, разработка нечеткой ЭВМ осуществляется снизу вверх, от элементной базы к архитектуре, которая сама еще до конца не разработана.

2. Создание вычислительных комплексов обработки нечеткой информации. По существу, основу подхода составляет разработка специализированных нечетких процессоров, предназначенных для использования в качестве сопроцессоров обычных ЭВМ.

Непосредственно для реализации нечетких алгоритмов и моделей предназначен ассоциативный параллельный процессор [Мелихов и др., 1981] — основа схем лингвистического терминала, с помощью которого оператор может взаимодействовать с ЭВМ, используя привычные для него качественные категории и оценки типа «много», «довольно низкий» и т. д. Кроме того, лингвистический термин может использоваться автономно как адаптивный нечеткий контроллер для управления технологическими процессами. Для реализации нечетких логических функций типа композиции используется ассоциативный параллельный процессор. В Японии также ведутся работы по созданию интегральных схем нечеткой логики, а также нечеткого процессора [Togai et al., 1986].

В данном разделе справочника приведено описание архитектуры и способов реализации нечетких алгоритмов вычислительным комплексом обработки нечеткой информации, который можно использовать для решения определенного круга задач нечеткой математики.

### Основные классы операций

Пусть  $X$  — произвольное непустое множество. Множество  $A = \{ \langle \mu_A(x) | x \rangle \}$ ,  $x \in X$ , называется нечетким подмножеством множества  $X$ , если каждый элемент множества  $A$  есть пара, на первом месте которой значенье функции  $\mu_A: X \rightarrow [0, 1]$  — функции принадлежности элементов из  $X$  множеству  $A$ , а на втором — элемент  $x \in X$ , для которого определена эта функция. Другими словами, при задании множества  $A$  каждому  $x \in X$  приписывается число  $0 \leq \mu_A(x) \leq 1$ , определяющее степень принадлежности этого элемента множеству  $A$ . Примем, что в множество  $A$  не включаются элементы  $\langle \mu_A(x) | x \rangle$ , для которых  $\mu_A(x) = 0$ .

Один из примеров нечеткого множества приведен основоположником теории, видным американским ученым Л. Заде. Допустим, что необходимо определить семантику понятия «высокий» человек. Для этого обратимся к процедуре экспертного опроса. Судя по рис. 5.57, степень соответствия роста 175 см понятию «высокий» равна 0,8 (т. е. можно сказать, что степень истинности высказывания «человек роста 175 см — это высокий человек» равна 0,8), а роста 200 см и выше 1,0. По существу, полученный график задает нам бесконечное нечеткое множество, описывающее понятие «высокий». Конечное нечеткое множество получается дискретизацией шкалы «Рост» с определением степеней соответствия в точках дискретизации и может выглядеть, например, так:

$$C = \{ \langle 0,1/150 \rangle, \langle 0,4/160 \rangle, \langle 0,7/170 \rangle, \langle 0,9/180 \rangle, \\ \langle 0,95/190 \rangle, \langle 1/120 \rangle \}.$$

Получив описание слова «высокий» в интерпретации «Рост», мы естественным образом подошли к понятию лингвистической переменной — это тройка  $\langle \alpha, T, X \rangle$ , где  $\alpha$  — название лингвистической переменной;  $T$  — термножество, т. е. множество словесных значений;  $X$  — базовое множество или базовая шкала. Если для описания роста человека использовать слова «маленький», «небольшой», «средний», «довольно высокий», «высокий», а для измерения использовать шкалу от 0 до 200 с шагом 50, то можно ввести лингвистическую переменную  $\langle \text{Рост}, \{ \text{«маленький»}, \dots, \text{«высокий»} \}, \{ 0, 50, \dots, 200 \} \rangle$ . Вербальные значения лингвистической переменной описываются нечеткими переменными, которые в общем виде представляются в виде тройки  $\langle \beta, C, X \rangle$ , где  $\beta$  — название нечеткой переменной;  $C$  — нечеткое множество, соответствующее  $\beta$ . Нечеткая переменная для значения «высокий» лингвистической переменной «Рост» есть тройка  $\langle \text{высокий}, C, X \rangle$ , где  $X = \{ 0, 10, 20, \dots, 200 \}$ .

Рассмотрим особенности выполнения операций над нечетким множеством. Условно можно выделить три класса наиболее часто используемых операций.

1. *Множественные операции* — все операции над нечеткими множествами  $A = \{ \langle \mu_A(x) | x \rangle \}$ ,  $B = \{ \langle \mu_B(x) | x \rangle \}$ ,  $x \in X$ , в результате которых строится новое нечеткое множество  $C = \{ \langle \mu_C(x) | x \rangle \}$ , где  $\mu_C(x) = \mu_A(x) \star \mu_B(x)$ , а  $\star$  обозначает нечеткую логическую операцию. Например, к множественным относится операция объединения нечетких множеств  $A \cup B = C = \{ \langle \mu_C(x) | x \rangle \}$ ,  $x \in X$ , где  $\mu_C(x) = \mu_A(x) \vee \mu_B(x)$ . Множественными являются также операции пересечения, разности, симметрической разности нечетких множеств. К частному типу множественных операций относится и операция дополнения. Как правило, такие операции используются при построении составного лингвистического высказывания, которое сводится к вычислению выражения типа

$$(\tilde{A} \cap \tilde{B}) \cup (\tilde{A} \cap \tilde{B}).$$



Рис. 5.57. Функция принадлежности понятия «высокий» (рост)

2. *Соотносительные операции* — это операции, результатом которых является некоторое число из замкнутого интервала  $[0, 1]$ , характеризующее соотношение между нечеткими множествами  $\tilde{A}$  и  $\tilde{B}$ . В общем виде такие операции описываются следующим образом:

$$\gamma(\tilde{A}, \tilde{B}) = \bigotimes_{x \in X} (\mu_{\tilde{A}}(x) * \mu_{\tilde{B}}(x)),$$

где  $\bigotimes$  и  $*$  — нечеткие логические операции. Примером соотносительной операции над  $\tilde{A}$  и  $\tilde{B}$  является операция определения степени нечеткого равенства  $\tilde{A}$  и  $\tilde{B}$ :

$$\gamma(\tilde{A}, \tilde{B}) = \bigotimes_{x \in X} (\mu_{\tilde{A}}(x) \leftrightarrow \mu_{\tilde{B}}(x)).$$

[Мелихов и др., 1981]. Соотносительными являются также операции вычисления нечеткого включения, нечеткой общности нечетких множеств. Операции этого класса часто используются при реализации нечетких ситуационных алгоритмов управления, например для идентификации конфликтных ситуаций [Мелихов и др., 1986а].

3. Операции *нечеткой арифметики*, которые, вообще говоря, имеют множественный характер, но наряду с выполнением нечетких логических операций предполагают и арифметические вычисления. В результате выполнения операций этого класса над  $\tilde{A}$  и  $\tilde{B}$  строится  $\tilde{C}$ , где  $\mu_{\tilde{C}}(z) = \bigvee_{z=x*y} (\mu_{\tilde{A}}(x) \& \mu_{\tilde{B}}(y))$ ,

а  $*$  — знак арифметической операции. Например, в результате операции расширенного сложения [Нечеткие, 1986]  $\tilde{A}$  и  $\tilde{B}$  строится  $\tilde{C} = \{ \langle \mu_{\tilde{C}}(z) | z \rangle \}$ , где  $\mu_{\tilde{C}}(z) = \bigvee_{z=x+y} (\mu_{\tilde{A}}(x) \& \mu_{\tilde{B}}(y))$ . (Отметим, что операции нечеткой арифметики

выполняются над нечеткими подмножествами множества действительных чисел.)

Операции указанных классов наряду с некоторыми особенностями выполнения обладают и общими чертами, главная из которых состоит в том, что при представлении нечетких множеств  $\tilde{A}$  и  $\tilde{B}$  в виде векторов степеней принадлежности все описанные выше операции сводятся к относительно независимому попарному выполнению нечетких логических операций (возможно, дополняемых арифметическими). Например, в случае определения объединения  $\tilde{A} = \{ \langle 0,1/1 \rangle, \langle 0,2/2 \rangle, \langle 0,5/3 \rangle \}$  и  $\tilde{B} = \{ \langle 0,4/1 \rangle, \langle 0,8/2 \rangle, \langle 0,3/3 \rangle \}$ , заданных на одном носителе  $X = \{1, 2, 3\}$ , наиболее информативно представление нечетких множеств в виде векторов  $\tilde{A} = (0,1; 0,2; 0,5)$  и  $\tilde{B} = (0,4; 0,8; 0,3)$ .

Определение вектора  $\tilde{C}$ , однозначно задающего  $\tilde{C}$ , получаемое в результате выполнения операции объединения, сводится к независимому попарному выполнению операции дизъюнкции над элементами векторов  $\tilde{A}$  и  $\tilde{B}$ . При использовании минимаксной трактовки нечетких логических операций [Заде, 1976]  $\tilde{C} = (0,4; 0,8; 0,5)$ , что соответствует нечеткому множеству  $\tilde{C} = \{ \langle 0,4/1 \rangle, \langle 0,8/2 \rangle, \langle 0,5/3 \rangle \}$ .

Итак, при векторном представлении нечеткого множества (практика показывает, что такое представление достаточно эффективно) наиболее предпочтительным вычислительным устройством для реализации операций над нечетким множеством является векторный процессор, в котором за счет разделения времени выполнения операций над парами элементов векторов степеней принадлежности можно существенно сократить время выполнения операций, в пределах до времени, сравнимого с затрачиваемым на обработку одной пары.

Значительное повышение скорости решения задачи достигается в параллельных вычислительных устройствах путем разбиения задачи на ряд относительно независимых подзадач, выполняемых на отдельных процессорах. При этом в качестве элементарных процессоров не обязательно использовать универсальные. Достаточно, определив класс задач, выявить необходимый набор элементарных операций и создавать специализированные элементарные процессоры с ограниченным набором операций, но либо с повышенной эффективностью их реализации, либо более простые, надежные и дешевые, чем универсальные.

Рассмотрим возможность использования параллельного векторного процессора (ПВП) для реализации операций над нечеткими множествами. С целью построения ПВП изучается класс задач, на который будет ориентирован ПВП, и определяется размерность  $N$  нечетких множеств, над которыми необходимо выполнять операции ( $N$  равно количеству элементов в самом длинном векторе степеней принадлежности). Выявляется набор элементарных операций, который должен выполнять элементарный процессор. Далее организуется линейка из процессоров, в которую по мере необходимости будут поступать векторные представления нечетких множеств и в которой будут выполняться элементарные операции, соответствующие решаемой задаче. В данном случае внимание ограничено только чисто вычислительным аспектом задачи. Способ засылки векторов степеней принадлежности в ПВП не рассматривается.

Для большинства практических приложений [Мелихов и др., 1979; Берштейн и др., 1983; Мелихов и др., 1986б; Нечеткие, 1986; Mamdani, 1977; Melikhov et al., 1987] характерно хранение некоторого набора эталонных нечетких множеств, который используется для преобразования и обработки входной и(или) выходной информации, также описываемой нечетким множеством. Как правило, набор эталонных нечетких множеств хранится в оперативной памяти. При использовании ПВП скорость реализации операций возрастает по сравнению с однопроцессорным вариантом, но этот выигрыш во времени совершенно поглощается временными затратами на последовательную поэлементную пересылку эталонных нечетких множеств в ПВП. Кроме того, значительно возрастают аппаратные затраты, так как кроме  $N$  элементарных процессоров, что уже сложнее, чем один универсальный или специализированный) в ПВП необходим управляющий процессор, организующий выборку данных из оперативной памяти, засылку их в элементарные процессоры, настройку и запуск элементарных процессоров для реализации требуемых операций и т. д. Таким образом, структура «ПВП — оперативная память» скорее всего не дает выигрыша в быстроте действия по сравнению с универсальной ЭВМ.

Основная причина неудачи — использование оперативной памяти и связанные с этим пересылки «оперативная память — ПВП». Заменяем пассивную память на активную, для чего в состав каждой ячейки оперативной памяти введем элементарный процессор. Такая память не может быть большой и должна программироваться по типу перепрограммируемого постоянного запоминающего устройства (ПЗУ). В совокупности ячейки ПЗУ составляют матричный процессор (МП), состоящий  $N \times M$  элементарных процессоров с внутренней памятью. При этом  $N$  равно мощности наиболее длинного векторного представления нечетких множеств для выбранного класса задач, а  $M$  определяется мощностью множества эталонных нечетких множеств. Вся информация, необходимая для обработки входных данных, хранится в элементарных процессорах. Входная информация поступает параллельно на все «линейки» МП. Однако при более или менее больших значениях  $M$  аппаратные затраты на реализацию МП (по крайней мере, на современном уровне развития технологии) поглощают все возможные выгоды от уменьшения времени выполнения операций над нечеткими множествами.

Итак, при разработке эффективных специализированных вычислительных устройств реализации операций над нечеткими множествами пока не могут ис-

пользоваться ни ПВП, ни МП. Нужен какой-то компромисс между возможным повышением быстродействия и аппаратными затратами. В качестве такого компромиссного варианта предлагается схема «элементарный процессор — полуактивная память», которая положена в основу вычислительного комплекса обработки нечеткой информации (ВКОНИ), разработанного в Таганрогском радиотехническом институте.

Понятие полуактивной памяти (ПАП) используется для обозначения структуры, организованной по типу МП, но отличающейся тем, что элементарные процессоры могут выполнять не все элементарные операции, необходимые для реализации характерных для выбранного круга задач операций над нечеткими множествами, а только часть из них, но наиболее часто используемую. В реализации ВКОНИ элементами ПАП выполняются операции типа концентрации и растяжения [Заде, 1976]. Выполнение остальных элементарных операций возлагается на универсальный или специализированный процессор, причем набор фиксирован и универсальный процессор может быть настроен на их выполнение на уровне микропрограмм [Куприянов и др., 1983], а специализированный процессор — аппаратно. Фиксированный набор операций нечеткой логики, на который настроен процессор, дает нам право называть его нечетким процессором (НП).

### Вычислительный комплекс обработки нечеткой информации

Структура ВКОНИ изображена на рис. 5.58.

Опытный образец ВКОНИ реализован как приставка к диалоговому вычислительному комплексу ДВК-2М, в котором в качестве управляющего процессора (УП) и оперативного запоминающего устройства (ОЗУ), используются процессор и ОЗУ ДВК-2М [Мелихов и др., 1986б; Баронец, 1986].

Управляющий процессор: интерфейс МПИ ОСТ 11.305.903-80; система команд микроЭВМ «Электроника-60»; тактовая частота 5 МГц; время выполнения команд 0,75—12 мкс.

Нечеткий процессор: интерфейс МПИ ОСТ 11.305.903—80; число команд 42 (реализованы основные операции нечеткой логики в различных базисах); тактовая частота 10 МГц, 16 внутренних регистров, адресное пространство, занимаемое на шине, 16 слов.

Полуактивная память: интерфейс МПИ ОСТ 11.305.903—80; адресное пространство, занимаемое на шине, 10 слов, объем ПАП — до 15 векторов степеней принадлежности НМ мощностью 101 элемент.

Элементная база: ИС серий К1801, К1804, К155, К589, К531, К556.

**Нечеткий процессор.** Для реализации операций над нечеткими множествами разработан специализированный процессор, который используется в ВКОНИ в качестве нечеткого.

Нечеткий процессор состоит из следующих основных блоков (рис. 5.59): блока связи с каналом (БСК) для приема и выдачи сигналов при обмене информацией с ОЗУ и другими внешними устройствами ВКОНИ, который включает формирователь управляющих сигналов (ФУС), регистр адреса (РА), дешифратор (ДША) и компаратор (КА) адреса, узел выдачи вектора прерываний (УВВП), регистр данных (РД);

операционного блока для обработки и хранения операндов (элементов нечеткого множества);

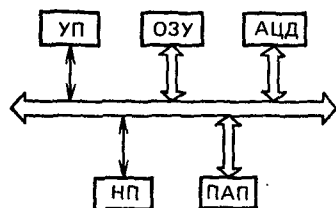


Рис. 5.58. Структура вычислительного комплекса обработки нечеткой информации

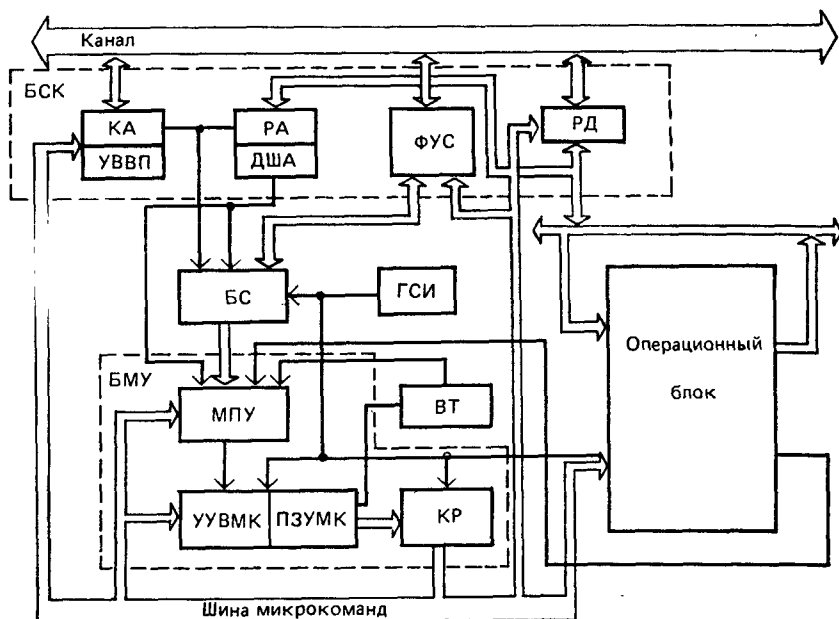


Рис. 5.59. Функциональная схема нечеткого процессора

блока микропрограммного управления (БМУ), состоящего из мультиплексора условий (МПУ), узла управления выборкой микрокоманд (УУВМК), ПЗУ микрокоманд (ПЗУМК), конвейерного регистра (КР), служащего для организации и координации работы всех узлов НП;

внутреннего таймера (ВТ) для формирования временных меток путем организации временных задержек между различными сигналами, максимальное время задержки составляет 256 тактов или 51,2 мкс;

блока синхронизации (БС) для временных привязок сигналов канала к работе БМУ и операционного блока;

генератора синхриомпульсов (ГСИ) для синхронизации работы всех блоков НП.

Процессор выполняет следующие операции над элементами нечеткого множества: чтение исходной информации из ОЗУ, запись результатов в ОЗУ, поиск элемента с максимальным (минимальным) значением в нечетком множестве. Помимо этого НП эффективно реализуются операции нечеткой логики в различных базах. Команды НП перечислены в табл. 5.3. Вместо операции \* (11—14) могут использоваться операции 7—10, а операция \* заменяется на операции 1—14.

В основу функционирования НП положен принцип микропрограммного управления, когда каждой операции соответствует определенная микропрограмма. Основной режим работы НП — режим прямого доступа к памяти, который также организуется микропрограммно.

**Полуактивная память.** В ячейках ПАП последовательно записаны векторные представления эталонных нечетких множеств. В ПАП осуществляется последовательный доступ к элементам векторов, поэтому главными элементами ПАП (рис. 5.60) являются ПЗУ элементов (ПЗУЭ), предназначенное для хранения элементов векторов степеней принадлежности нечетких множеств, и ПЗУ



Система команд нечеткого процессора

№ п/п	Операция	Функция
1	$r := \bar{a}$	Пересылка
2	$r := 1 - \bar{a}$	Дополнение
3	$r := \max(\bar{a}, b)$	Дизъюнкция
4	$r := \min(\bar{a}, b)$	Конъюнкция
5	$r := \min(1, \bar{a} + b)$	Ограниченная сумма
6	$r := \max(0, \bar{a} + b - 1)$	Ограниченное произведение
7	$r := \max(1 - \bar{a}, b)$	Импликация
8	$r := \min(1, 1 - \bar{a} + b)$	→—
9	$r := \begin{cases} 1, & \text{если } \bar{a} \leq \bar{b} \\ 0, & \text{если } \bar{a} > \bar{b} \end{cases}$	→—
10	$r := \begin{cases} 1, & \text{если } \bar{a} \leq \bar{b} \\ b, & \text{если } \bar{a} > \bar{b} \end{cases}$	→—
11—14	$r := \min(\bar{a} * b, b * \bar{a})$	Эквивалентность
15—28	$\max\{\bar{a}_i \otimes b_j\}$	Взятие максимума среди элементов нечеткого множества, которое получается в результате выполнения операции $\times$ над множествами $\bar{A}$ и $B$
29—42	$\min\{\bar{a}_i \otimes b_j\}$	Взятие минимума

степеней (ПЗУС) элементов, предназначенное для хранения до 8 степеней элементов «векторных» представлений нечетких множеств. В данной реализации ПАП может выполнять только две операции над четкими множествами: концентриацию и растяжение, которые сводятся к возведению элементов векторов в соответствующие степени, например в степень 2 или 1/2. В состав ПАП входят также следующие блоки: ФУС при обмене с каналом; РА; ДША; РД для выдачи элементов термов в канал; регистры номеров термов РН0—РН7, элементы которых должны быть считаны в канал; счетчик числа элементов (СчЭ) для организации последовательного к ним доступа; регистр количества термов (РКТ) для определения числа считываемых термов; регистр текущего номера (РТ) для указания РН, содержащего номер терма, элементы которого в данный момент считываются; схема управления (УРТ) регистром РТ; мультиплексор МП1 для выбора одного из РН; формирователь адреса элемента (ФАЭ); формирователь адреса степени (ФАС); мультиплексор МП2 для выдачи информации из ПЗУЭ или ПЗУС в РД.

**Реализация нечетких алгоритмов специализированным процессором ВКОНИ.** Общая организация данных, необходимых для обработки нечеткой информации, и основные операции, выполняемые в ходе реализации нечетких алгоритмов, типа «ситуация — действие» [Mamdani, 1977; Melikhov et al., 1987], которые являются частным случаем ситуационных сетей [Мелихов и др., 1986а], следующие.

Управление объектом моделируется в ВКОНИ набором эталонных нечетких ситуаций, каждой из которых ставится в соответствие управляющее решение, возможно, также нечетко формализованное. Эталонная ситуация описывается нечетким множеством второго уровня [Заде, 1976] на множестве признаков  $Y = \{y_1, y_2, \dots, y_p\}$  объекта. В качестве признаков могут использоваться, например, скорость движения, высота полета, погодные условия. Выбор признаков определяется целью управления объектом. Каждый признак  $y_i, i \in I = \{1, 2, \dots, p\}$ , описывается соответствующей лингвистической переменной

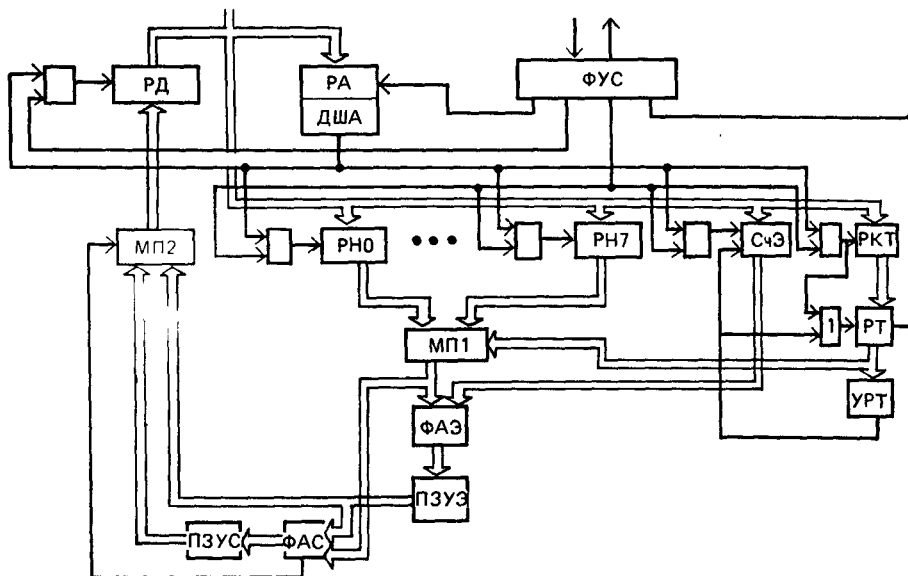


Рис. 5.60. Функциональная схема полуактивной памяти

$\langle y_i, T_i, D_i \rangle$ , где  $T_i = \{T_1^i, T_2^i, \dots, T_{m_i}^i\}$  — ее терм-множество (набор вебральных значений типа «небольшая», «довольно высоко», «неплохая» и т. д.);  $m_i$  — количество значений;  $D_i$  — базовое множество признака  $y_i$ , т. е. конкретная количественная шкала изменения числовых значений признака. Выбор  $T_i$  и  $D_i$  для признака  $y_i$  определяется целями управления.

Для описания термов  $T_j^i$ ,  $j \in J = \{1, 2, \dots, m_i\}$ , используются нечеткие переменные  $\langle T_j^i, D_i, C_j^i \rangle$ , где  $C_j^i$  — нечеткое множество в базовом множестве  $D_i$ , определяющее значение  $T_j^i$ :

$$\tilde{c}_j^i = \{ \langle \mu_{C_j^i}(d) \mid d \rangle \}, \quad d \in D_i.$$

Тогда нечеткая ситуация, описывающая некоторое состояние объекта управления, представляется в виде нечеткого множества второго уровня

$$\tilde{S} = \{ \langle \mu_S(y_i) \mid y_i \rangle \}, \quad y_i \in Y,$$

$$\text{где } \mu_S(y_i) = \{ \langle \mu_{S(y_i)}(T_j^i) \mid T_j^i \rangle \}, \quad i \in I, j \in J.$$

Набор эталонных ситуаций более или менее полно (полнота зависит от подготовленности эксперта, используемого для моделирования управления объектом) отражает возможные ситуации, возникающие на объекте управления. Алгоритм «ситуация — действие» состоит в поиске эталонной ситуации, в которой находится объект управления, и выдаче соответствующего ей управляющего решения. Поиск осуществляется методом ближайшего соседа. Для этого описание текущего состояния объекта управления, представленное в виде входной нечеткой ситуации, сравнивается со всеми эталонными ситуациями. Объект считается находящимся в эталонной ситуации, в некотором смысле наиболее близкой к входной ситуации. В качестве меры близости в ВКОНИ используются степени нечеткого равенства и нечеткого включения [Мелихов, 1986б]. Вычисление их состоит в выполнении операций нечеткой эквивалентности или

нечеткого включения над элементами векторов значений степеней принадлежности элементов нечетких множеств с последующим определением минимального результата операции.

Если число эталонных нечетких ситуаций не превышает 15, то нечеткий алгоритм «ситуация — действие» реализуется ВКОНИ наиболее эффективно, примерно на три порядка быстрее, чем на мини-ЭВМ типа «Электроника 100/25», примененной в качестве контрольной [Берштейн, 1983]. В этом случае эталонные ситуации заносятся непосредственно в ПАП, поиск текущей эталонной ситуации производится квазипараллельно по всем эталонным ситуациям. Для этого НП из ПАП пересылаются сначала первые элементы всех эталонных ситуаций, затем вторые и т. д. При этом выполняются необходимые нечеткие логические операции над первым элементом входной ситуации и всеми первыми элементами эталонных ситуаций, затем над вторым элементом входной и всеми вторыми элементами эталонных и т. д. Результаты выполнения операций заносятся в соответствующие регистры, в которых определяется минимум между их содержимым и заносимым числом. После «просмотра» всех элементов эталонных ситуаций в регистрах хранятся степени схождения входной нечеткой ситуации с эталонными. Регистр с максимальной степенью соответствует эталонной ситуации, в которой находится объект управления.

Если число эталонных ситуаций более 15, алгоритм «ситуация — действие» реализуется по-другому. В ПАП хранятся терм-множества лингвистических переменных, описывающих признаки ситуации, точнее, в ПАП заносятся векторы степеней принадлежности нечетких множеств нечетких переменных, соответствующих вербальным значениям признаков. Для повышения эффективности использования объема ПАП туда можно заносить нечеткие множества на универсальной шкале значений признаков [Ежкова и др., 1977]. При этом переход от предметной шкалы к универсальной и наоборот производится управляющим процессором ВКОНИ. Теперь ПАП используется для идентификации значений признаков входных нечетких ситуаций, а также для решения задач аппроксимации и интерпретации лингвистической переменной. Описания эталонных ситуаций хранятся в ОЗУ. Поиск текущей эталонной ситуации осуществляется последовательным сравнением идентифицированного посредством использования ПАП описания входной нечеткой ситуации со всеми эталонными ситуациями, находящимися в ОЗУ. В этом случае получается выигрыш на один-два порядка по сравнению с контрольной ЭВМ. Это достигается благодаря использованию НП для выполнения необходимых операций нечеткой логики, а также ПАП для идентификации входной ситуации, аппроксимации и интерпретации входных результатов.

**Класс решаемых задач и возможные применения.** Наличие в составе ВКОНИ НП, обеспечивающего выполнение нечетких логических операций в любых базисах, обуславливает возможность его применения для решения широкого класса задач нечеткой математики. Помимо указанных выше, с помощью ВКОНИ можно довольно эффективно решать следующие задачи:

- нечеткие логические уравнения;
- задачи нечеткой арифметики;
- выполнение операций над нечеткими отношениями;
- приближенный логический вывод на основе нечеткой логики;
- построение нечетких классификаций и кластеризаций;
- нечеткое распознавание по признакам;
- лингвистическая аппроксимация;
- нечеткая оптимизация;
- нечеткие игры.

Возможны следующие практические применения ВКОНИ:

нечеткий контроллер — управление технологическими и организационными процессами на основе плохо определенной (в традиционном математическом смысле) информации;

лингвистический терминал — обеспечение общения с пользователем на языке, приближенном к естественному;

персональное советующее устройство — при реализации на ВКНИ персональной экспертной системы с нечеткой логикой может использоваться для выдачи советов по поведению в трудноформализуемых ситуациях в задаваемой пользователем предметной области.

## Глава 6.

# Специализированные процессоры для языков высокого уровня

## 6.1. Лисп-процессоры

А. Н. Мямлин, А. Г. Рубин, В. К. Смирнов

Лисп считается в мире общепринятым языком программирования для задач искусственного интеллекта [Уинстон, 1980, 1987; Нильсон, 1985; Boley, 1980] (см. § 1.3). Область применения языка Лисп чрезвычайно широка (символьная обработка, в том числе аналитические преобразования, конструирование компиляторов, построение САПР и др.). В последнее время его называют языком «больших» систем, поскольку на нем реализованы крупные программные комплексы.

Существенные особенности языка Лисп — его функциональная природа, рекурсивные списковые структуры, специфика распределения памяти, эквивалентность представления программ и данных — определяют своеобразие его применения. Основные функции Лиспа предназначены для установления логических связей между элементами данных в памяти ЭВМ, к которым производится обращение. Такой вид обработки данных хорошо подходит для построения сложных взаимосвязей, подобных тем, которые требуются для понимания естественного языка, работы реляционных баз данных, проектирования интегральных микросхем и функционирования информационно-поисковых систем. Память в Лиспе запрашивается динамически и распределяется всякий раз, когда это необходимо во время выполнения. Это особенно важно для задач искусственного интеллекта, в которых данные часто генерируются программой и могут иметь непредсказуемую заранее длину. Списковая структура Лиспа ставит его в особое положение при решении задач искусственного интеллекта, поскольку многие из них могут быть изоморфно представлены в виде деревьев.

Среди многочисленных диалектов Лиспа (табл. 6.1) *Common Lisp* рассматривается как основной претендент на стандарт языка. Он вобрал в себя наиболее важные свойства предшествовавших ему диалектов Лиспа [Уинстон, 1977; Steele, 1982, 1984]:

средства создания структур с процедурами доступа к отдельным полям записей;

оператор присваивания, позволяющий работать с переменными, массивами, структурами и свойствами;

механизм построения сложных выражений по образцу;

макросредства, позволяющие пользователю работать в собственном синтаксисе;

различные способы задания значений аргументов (в том числе по умолчанию и с помощью ключевых слов);

потоковую систему ввода-вывода информации.

С целью повышения эффективности работы с языком Лисп в разных странах ведутся интенсивные исследования в области разработки специализированных процессоров для применения их в интеллектуальных системах. Основными

Наиболее распространенные диалекты языка ЛИСП

Диалект	ЭВМ	Поддерживающая организация
MacLisp	PDP-10, Multics, DEC VAX, IBM PC, CADR, LAMBDA	МТИ
InterLisp	XEROX 1100	Научно-исследовательский центр фирмы XEROX; фирма Bolt, Beran- lek & Newman
Portable Standard Lisp (PSL)	IBM 360/370, PDP-10, B-1700/6700, Z-80, DEC VAX, MC68000, CDC 6600/7600, Univac-1108, Cray-1	Университет шт. Юта (лаборато- рия символьных вычислений); Стэнфордский ун-т
Zetalisp	DEC VAX, LAMBDA, Symbolics 3600, Explorer	Фирмы Symbolics Inc., Lisp Machi- nes Inc.
Franz Lisp	DEC VAX	Фирма DEC
Common Lisp	IBM PC, DEC VAX, MC68000, IPSC-MX, HP-9000, XEROX 1100, Macintosh, LAMBDA, Symbolics 3600, Explorer, ELIS	Фирмы Symbolics, LMI, DEC, XEROX, TI, Hewlett-Packard. Университеты и институты: МТИ, Карнеги-Меллона, Ливерморская лаборатория, Йельский, Bell. Lab. и др.

направлениями в области разработки современных Лисп-процессоров являются:

- аппаратная база реализации языка Лисп;
- архитектурные решения, применяемые в современных Лисп-системах;
- структурные и функциональные особенности специализированных Лисп-процессоров.

#### Развитие аппаратной базы реализации языка Лисп

Традиционная архитектура ЭВМ плохо приспособлена для эффективной реализации языка Лисп, поэтому разработка специализированных архитектур ЭВМ, предназначенных для поддержки языка Лисп, — специализированных Лисп-машин [Boley, 1980; Greeger, 1983; Маршалл, 1980] — стала актуальной задачей.

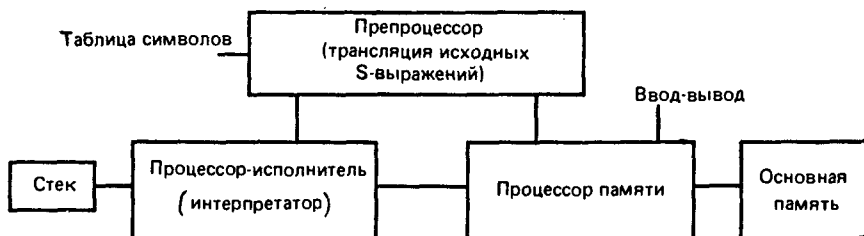


Рис. 6.1. Схема Лисп-системы CADM

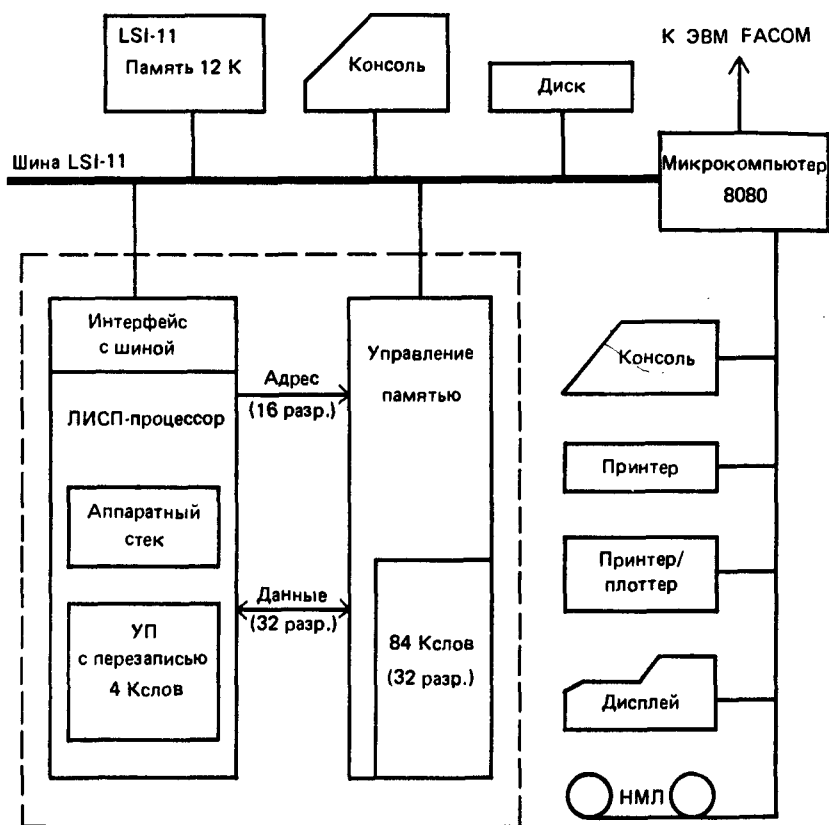


Рис. 6.2. Схема Лисп-машины (Kobe University) .

Экспериментальные Лисп-машины создавались на базе микропрограммного процессора большой или малой универсальной ЭВМ традиционной архитектуры, но с управляющей памятью и возможностью перезаписи. Для Лиспа специально разрабатывался лишь микропрограммный интерпретатор языка [Griss, et al., 1977; Deutsch, 1978; Yamamoto, 1981]. Таким образом, архитектура Лисп-машины может быть представлена как гибридная, которая выполняет и обычную поддержку линейной памяти со счетчиком команд и указателем стека, и организацию связанных списков, характерных для Лиспа.

При таком подходе специализированная микропрограммная поддержка и проблемно-ориентированный набор команд не позволяют преодолеть архитектурные недостатки аппаратной части, не учитывающей специфики языка Лисп. Использование только микропрограммной поддержки оказывается недостаточным для получения действительно эффективного варианта Лисп-системы. Для этого необходимо наиболее полно отразить те особенности Лиспа в аппаратуре, которые определяют выразительную мощност и другие преимущества языка, и компенсировать те особенности, которые вызывают трудности при программной и микропрограммной реализации или приводят к заметной потере эффективности.

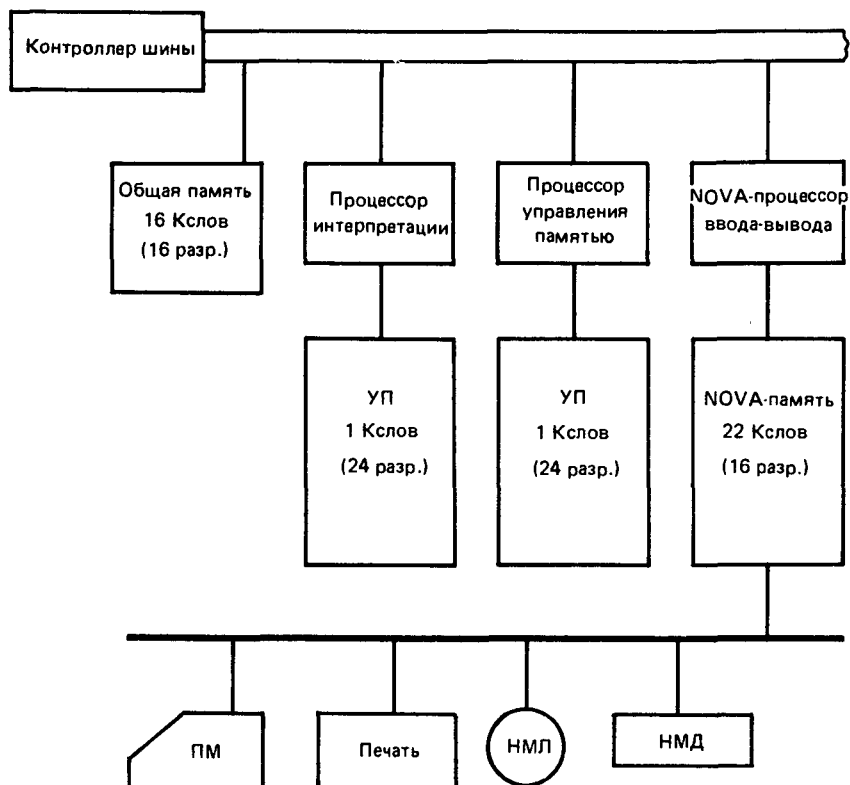


Рис. 6.3. Схема Лисп-машины (Keio University)

Переходный период от микропрограммных реализаций языка Лисп на универсальных процессорах к специализированным Лисп-процессорам выразился в погружении в аппаратуру лишь отдельных наиболее существенных архитектурных особенностей языка при сохранении основной части алгоритмов в микрокоде. В частности, в различных проектах Лисп-процессоров проводились исследования возможности аппаратной реализации таких ключевых элементов, как:

- работа с полями битов и тегам;
- извлечение полей записи, их маскирование и сдвиг;
- стек и операции работы со стеком;
- буферизация фрагментов стека в быстрой памяти;
- организация дополнительной быстрой памяти (кэш-память);
- использование буфера машинных инструкций и схемы опережающей загрузки последних;
- использование дешифратора инструкций и памяти адресов, обеспечивающей переход к микропрограммам операций;
- выделение группы дополнительных регистров;
- применение хэш-таблиц;
- организация специального арифметического блока, блока умножения;
- реализация сборщика мусора и средств для его поддержки;

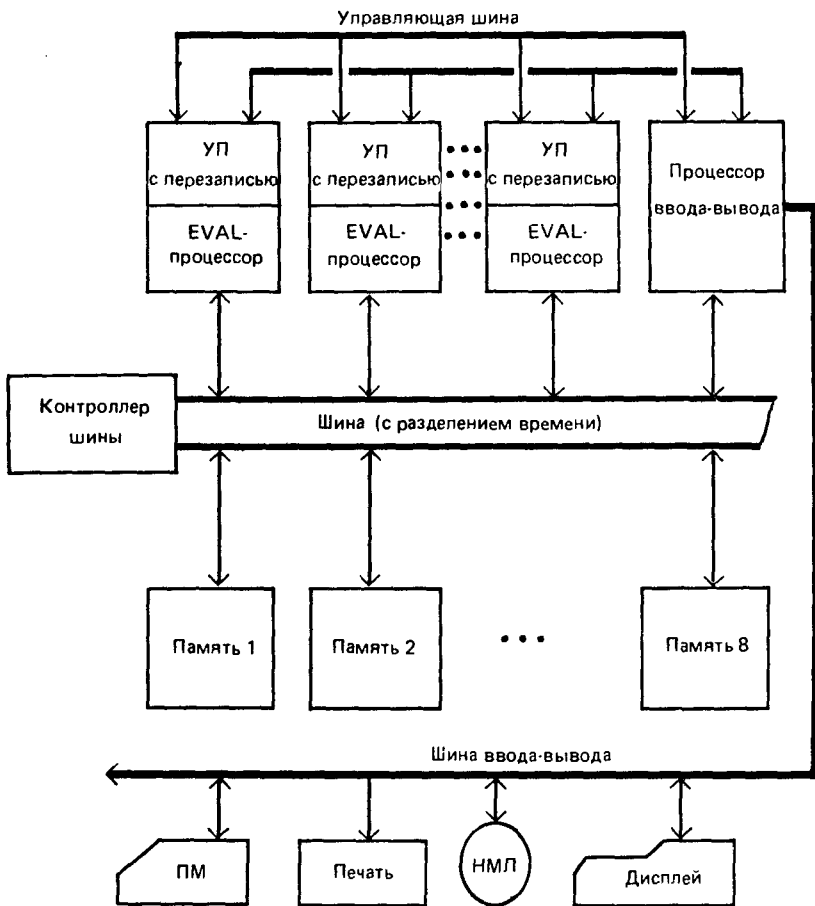


Рис. 6.4. Схема машины EVLIS (Osaka University)

реализация виртуальной страничной памяти;  
 организация интерфейса ввода-вывода, интерфейса с шиной или адаптера;  
 использование схемы обработки прерываний;  
 построение схемы диагностики и организация средств обеспечения отказо-  
 устойчивости.

Развитие технологии микропрограммирования, появление надежных и доступ-  
 ных микропроцессоров, совершенствование аппаратных средств и резкое уде-  
 шевление запоминающих устройств позволило создать производительные и  
 сравнительно дешевые Лисп-машины [Мямлин и др., 1988].

Уже в ранних реализациях экспериментальных Лисп-машины рассматрива-  
 лись варианты мультимикропроцессорной архитектуры. Например, проект Лисп-ма-  
 шины CADM (рис. 6.1), предусматривающий организацию секций препроцесси-  
 рования (трансляции), микропрограммной интерпретации и управления памятью,  
 реализован на базе трех микропроцессоров фирмы Intel. Дешевые, миниатюр-  
 ные, надежные микропроцессоры связаны через двунаправленную шину. Сис-  
 тема допускает интерпретацию Лисп-программ параллельно с выполнением фо-



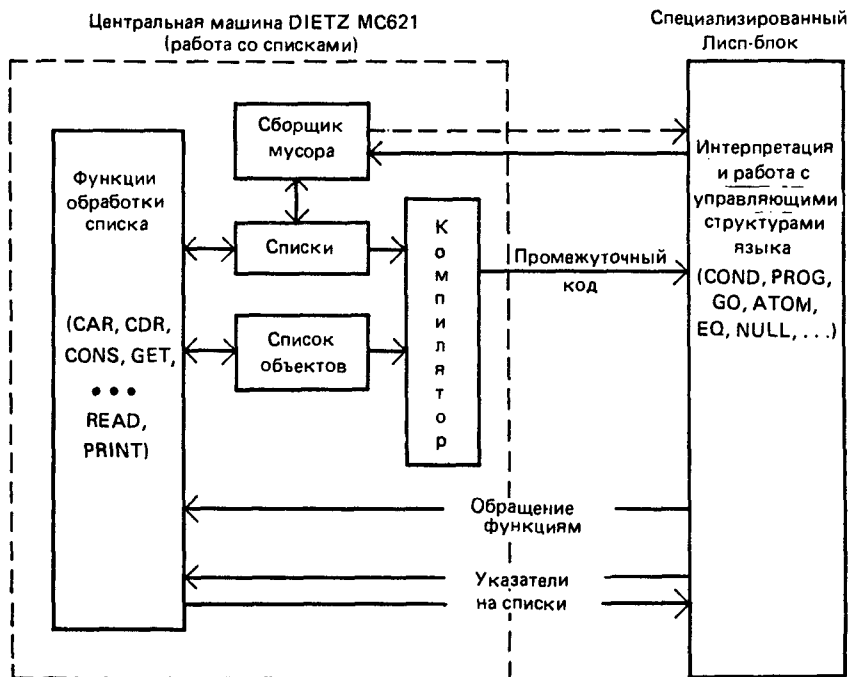


Рис. 6.5. Схема микропрограммной Лисп-машины

новой задачи «сборки мусора», благодаря мультипроцессированию и конвейерной обработке преодолеваются свойственные микропроцессорам ограничения в скорости.

Проект КОБЕ Lisp (рис. 6.2) базируется на четырех секционных микропроцессорах AM2900 с дополнительной аппаратной поддержкой. Через микрокомпьютер с микропроцессором 8080 осуществляется связь Лисп-процессора с универсальной ЭВМ типа FACOM 230-38, а микрокомпьютер DEC LSI-11 выполняет инициализацию, обслуживание, загрузку программ и операции ввода-вывода Лисп-процессора.

Экспериментальный проект мультипроцессорной Лисп-системы KEIO Lisp (рис. 6.3) реализован на базе двух специализированных процессоров, предназначенных для интерпретации и управления памятью, и мини-компьютера NOVA, играющего в системе роль процессора ввода-вывода. Функционирование всех трех компонентов осуществляется параллельно и независимо друг от друга.

Мультипроцессорная Лисп-система EVLIS (рис. 6.4) построена с использованием четырех секционных микропроцессоров 13000 фирмы Intel и процессора ввода-вывода. Проект интересен тем, что в отличие от традиционного подхода, при котором параллельно с интерпретацией осуществляется сборка мусора и ввод-вывод, в указанной системе на отдельных процессорах параллельно выполняются аргументы Лисп-функции EVLIS.

Еще одна экспериментальная Лисп-машина (рис. 6.5) представляет собой двухпроцессорную систему, состоящую из центральной 8-разрядной ЭВМ DIETZ MC621, предназначенной для компиляции, обработки списков сборки мусора, и присоединенного к ней специализированного микропрограммного устройства для интерпретации и работы с управляющими структурами языка.

## Современные Лисп-системы

В течение достаточно продолжительного времени в области искусственного интеллекта использовалась универсальная ЭВМ PDP-10 (DEC-10) или KA-10, которая во многих публикациях до настоящего времени служит некой точкой отсчета при сопоставлении характеристик появляющихся Лисп-машин. До разработки в Массачусетском технологическом институте (МТИ) в 1970 г. первой Лисп-машины CONS (так называемой Лисп-машины Гринблатта) все реализации Лиспа осуществлялись на универсальных процессорах, архитектура которых накладывала существенные ограничения на выполнение Лисп-программ. Цель проекта CONS состояла в создании машины, более приспособленной для выполнения Лисп-программ, чем KA-10. Проект оказал большое влияние на последующие разработки специализированных Лисп-процессоров. CONS представляла собой небольшой специализированный микропрограммный Лисп-процессор, требующий поддержки универсального центрального процессора.

Пробором первых коммерческих Лисп-машин основных фирм-производителей Symbolics Inc., Lisp Machines Inc. (LMI) и Texas Instruments Inc. (TI) стала машина CADR, созданная в МТИ в 1978 г. К началу промышленного выпуска Лисп-машин в МТИ было изготовлено около 20 таких компьютеров, два из которых были переданы в Научно-исследовательский центр фирмы Хетгох, где они были подключены к одной из наиболее популярных в США локальных сетей этой фирмы Ethernet. Лисп-машина CADR реализована по принципу микропрограммной стековой машины (с аппаратным стеком). Основной цикл машины 180 нс, емкость оперативной памяти 128 Кслов (32-разрядных) и памяти управления 16 Кслов (при длине микрокоманды 48 разрядов). Реализация Лиспа ориентирована на компиляцию, в каждом слове по две 16-разрядные команды. В качестве внешних устройств использовались диск и дисплей. Машина CADR более совершенна, чем ее предшественница, и на задачах, связанных с аналитическими вычислениями, решаемых с использованием системы Macsyma, имела в 3—4 раза более высокую производительность, чем CONS [Greeger, 1983].

### Лисп-процессоры фирмы Symbolics

Первыми фирмами, наладившими промышленный выпуск специализированных ЭВМ, реализующих Лисп, были американские фирмы LMI и Symbolics Inc. Эти фирмы образованы в конце 70-х годов специалистами, участвовавшими ранее в проекте CADR и поставившими себе целью разработку и создание коммерческих Лисп-машин. Вариант первой промышленной Лисп-машины фирмы LMI практически повторял машину CADR лишь с незначительными модернизациями. LMI оказалась первой фирмой, предложившей пользователям (к середине 1981 г.) промышленную Лисп-машину. Первая промышленная Лисп-машина фирмы Symbolics — LM-2 (табл. 6.2) также почти копировала машину CADR и была поставлена на рынок в конце 1981 г.

Машины LAMBDA (фирмы LMI) и Symbolics 3600 (табл. 6.2) представляют второе поколение Лисп-машин, построенных уже на основе оригинальной архитектуры и современной элементной базе.

Лисп-машина фирмы Symbolics — совершенно автономная ЭВМ с собственным дисплеем и диском. Семейство Symbolics 3600 имеет стекоориентированную архитектуру с высокоскоростной буферизацией фрагментов стека. В процессе счета задачи осуществляется проверка типов данных, обнаруживаются инициализированные переменные и ошибки связывания с помощью специальной аппаратуры. Высокая производительность машины достигается за счет введения дополнительных шин передачи данных, совмещения выполнения команд и реализации оптимального их набора.

Семейство Symbolics 3600 включает несколько программно-совместимых моделей, отличающихся емкостью оперативной памяти, особенностями аппаратуры, обеспечивающими различную производительность, составом и качеством

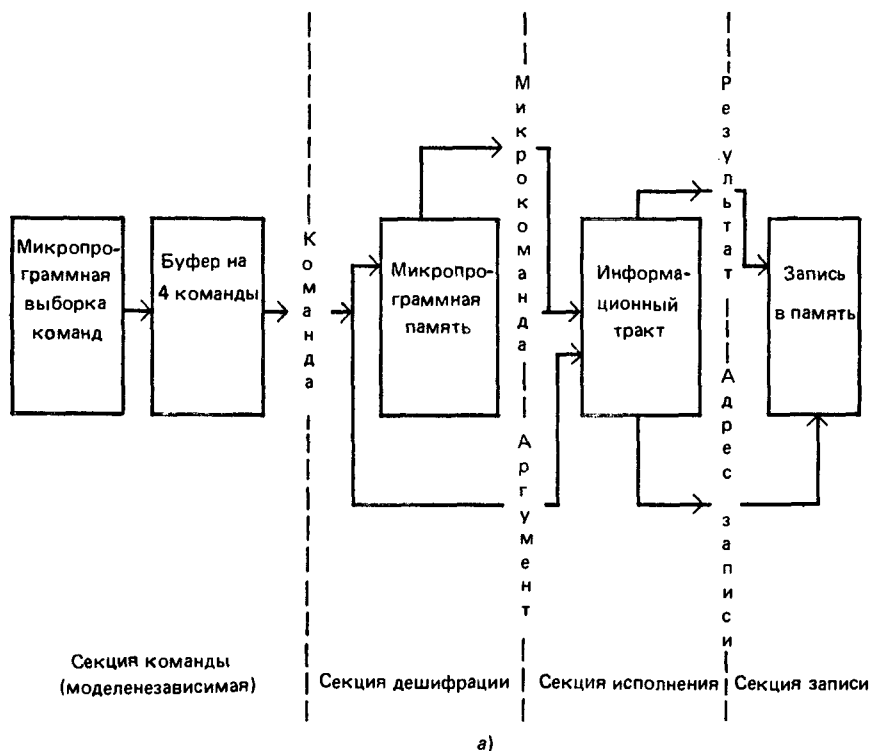
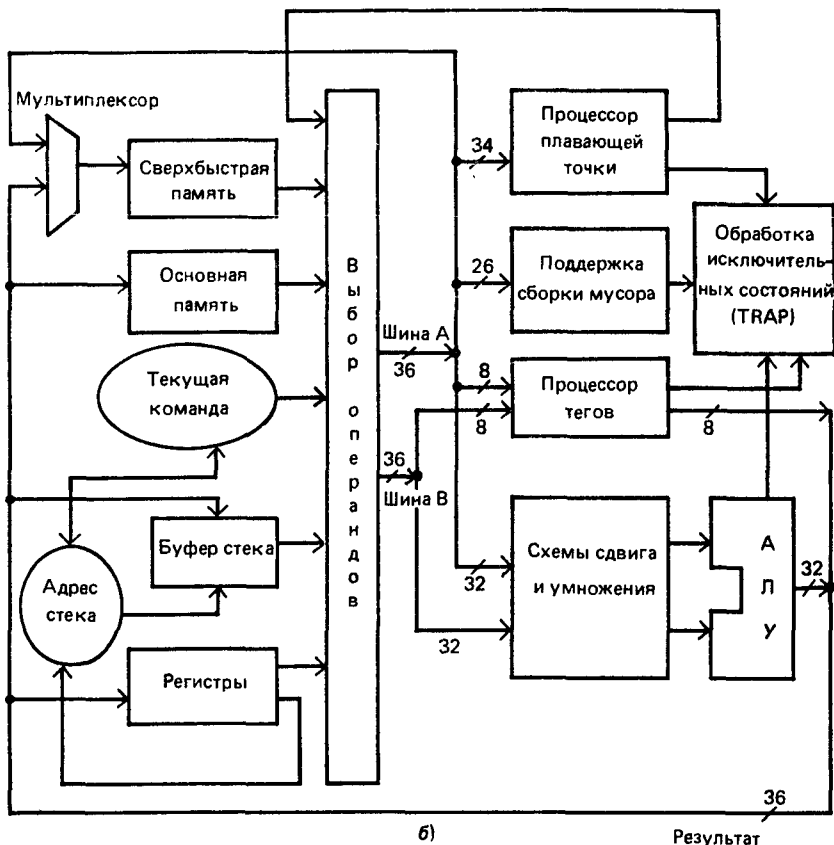


Рис. 6.6. Магистраль обработки команд (а) и секции исполнения и записи (б) модели Symbolics

внешних устройств. Например, старшие модели 3675 и 3620 имеют аппаратный блок выборки команд и кэш-память, более ранняя модель Symbolics 3640 — блок микропрограммной выборки команд и кэш-память (буфер) на четыре команды. В машинах серии 3600 (рис. 6.6,а) добавление тега к слову памяти сопровождалось увеличением разрядности (до 36 разрядов), что позволило увеличить адресное пространство (32-разрядные адреса) и работать с 32-разрядными данными [Moop, 1985, 1987]. При выполнении Лисп-программ младшие модели Symbolics 3600 показали в 2—4 раза более высокую производительность, чем машина VAX 11/780 фирмы DEC, имеющая приблизительно тот же основной цикл в 200 нс, и на простых тестах оказались приблизительно в 3 раза быстрее, чем CADR МТИ [Маршалл, 1980; Moop, 1987; Диринг, 1987].

В настоящее время фирма выпускает модели 3610, 3620, 3650 и 3675. В моделях 3620 и 3650 процессор, выполненный на логических матрицах, занимает одну плату. К 1989 г. фирма запланировала большой проект на 16 млн. дол. по созданию 40-разрядной СБИС более высокой производительности [Catier, 1987].

**Лисп-процессоры, включенные в традиционную вычислительную среду.** Другое распространенное архитектурное решение Лисп-системы предполагает использование специализированного Лисп-процессора в качестве сопроцессора универсальной ЭВМ. Причем к универсальной ЭВМ в различных проектах предъявляются самые разные требования — от поддержки некоторых функций



Лисп-процессора (например, сборки мусора) и реализации средств инициализации, ввода-вывода, загрузки Лисп-процессора (с предоставлением ему внешних устройств) до тестирования Лисп-процессора и обслуживания его потребностей универсальной операционной системой центральной ЭВМ, предоставления ему средств управления виртуальной памятью и пр. [Yamamoto, 1981; Greeger, 1983; Puttkamer, 1983]. Примером такого сопроцессора является Лисп-машина ALPHA японской фирмы Fujitsu, обеспечивающая повышение производительности в 5—6 раз по сравнению с программной Лисп-системой, реализованной на машине FACOM с сопоставимой аппаратурой [Hayashi et al., 1984].

Машина ALPHA (рис. 6.7) представляет собой микропрограммный Лисп-процессор, подключенный через блок-мультиплексный канал к универсальной ЭВМ FACOM, не имеет своих устройств ввода-вывода и доступна только с терминала центральной ЭВМ. К последней может быть подключено много процессоров ALPHA, и таким образом многочисленные пользователи системы разделения времени (TSS) на ЭВМ FACOM могут эффективно выполнять обработку символьной информации. ALPHA имеет аппаратный стек и схему проверки типов данных, схему опережающей выборки команд. Наряду с оборудованием фирма Fujitsu предлагает своим потребителям также инструментальные средства для разработки экспертных систем (пакет Eshell).

## Характеристики современных промышленных Лисп-машин

Модель (фирма)	Год выпуска	Цикл, нс	Емкость ОЗУ, Мслов	Емкость УП, Кслов	Емкость ВЗУ, Мбайт	Внешние устройства	Язык	Стоимость, тыс. дол.
LM-2 (Symbolics Inc., США)	1982	—	0,2 (32-разр.)	12 (48-разр.)	80	Черно-белый дисплей	MacLisp	80
LAMBDA (LMI, США)	1983	—	0,5—1 (32-разр.)	16 (64-разр.)	470 (через Nubus, Multibus)	Графический дисплей (800×1024 точки)	LM-LISP, MacLisp, Zeta-lisp, Common Lisp, C, Prolog, Pascal, Fortran,	52—140
Symbolics 3600 (Symbolics Inc., США)	1983—1986	200	0,5—6 (36-разр.)	16 (112-разр.)	169	Черно-белый дисплей	Zetalisp, PSL, Common Lisp, Prolog, Fortran, Ada	50—84
ALPHA (FUJITSU, Япония)	1983	160	2 (32-разр.)	16 (48-разр.)	Ввод-вывод через ЭВМ FACOM	Связь с ЭВМ FACOM через адаптер и байт-мультитиплексный канал	Utilisp (MacLisp), Prolog	—
Explorer (TI, США)	1984	—	4 (32-разр.)	16 (56-разр.)	112 (через Nubus)	Цветной графический дисплей (808×1024 точки)	Zetalisp, Common Lisp, Cobol, Pascal, Fortran	49—99
XEROX 1100 (Xerox Corp., США)	1982—1986	—	0,5 (16-разр.)	—	10—80 (и гибкие по 1 Мбайт)	Черно-белый или цветной графический дисплей	Interlisp, Smalltalk, Common Lisp, Cobol, Basic, Fortran, Pascal	10—16 и выше
ELIS (NTT, Япония)	1987	180	16 (64-разр.)	64 (64-разр.)	540	Связь с ВЛУ через процессор ввода-вывода MC68010	TAO, Common Lisp, Smalltalk, Prolog	—
SM45000 (IBM, США)	1987	—	4 (40-разр.)	—	—	Связь с ВЛУ через процессор ввода-вывода 80286	Common Lisp, Prolog	39—45

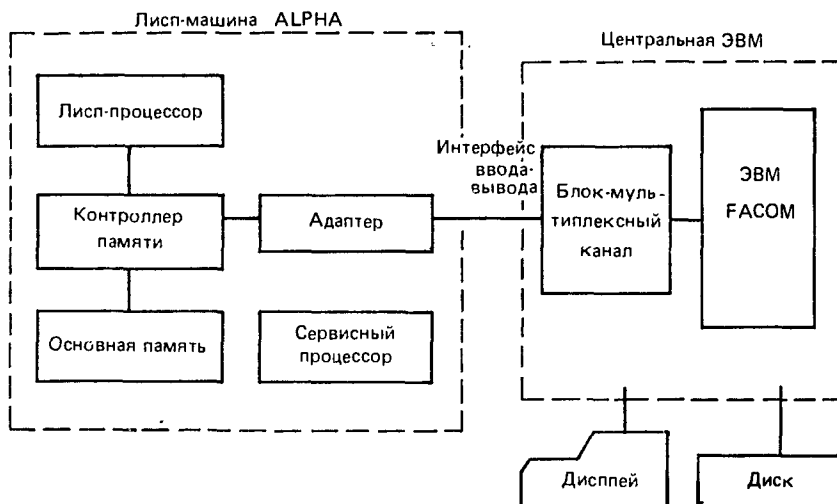


Рис. 6.7. Схема Лисп-системы ALPHA

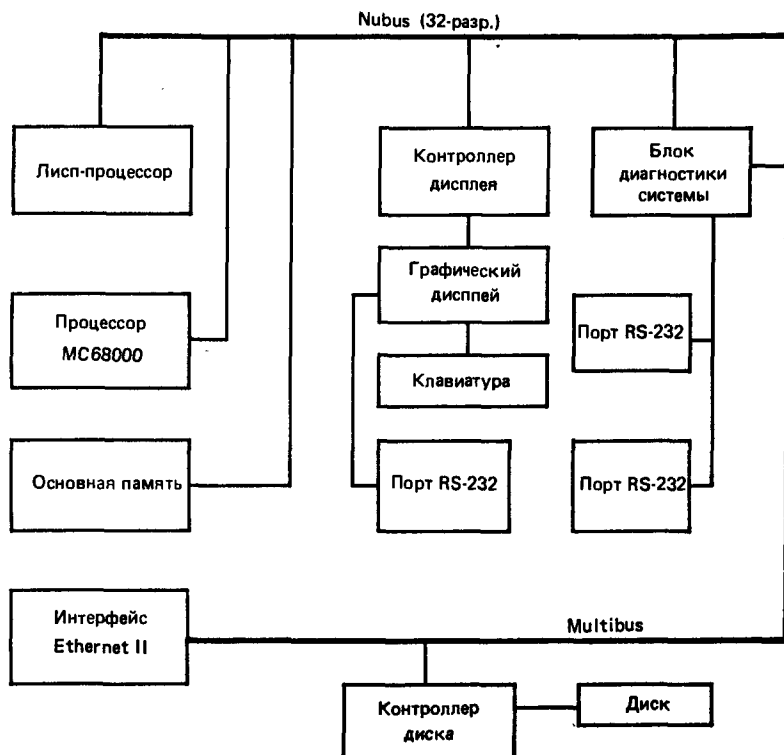


Рис. 6.8. Схема Лисп-машины LAMBDA фирмы LMI

В традиционных архитектурах мультипроцессорных систем все вычислительные мощности концентрировались вокруг одного центрального процессора. В системе LAMBDA фирмы LMI центром системы является шина NuBus, осуществляющая взаимодействие между отдельными устройствами, присоединенными к ней (рис. 6.8). NuBus-шина реализована как 32-разрядная с максимальной скоростью передачи 37,5 Мбайт/с, что позволяет поддерживать мультипроцессорную систему и обеспечивает архитектурную гибкость. Благодаря подключению к шине NuBus машина LAMBDA может комплексоваться практически в двухпроцессорную систему, объединяющую специализированный Лисп-процессор и универсальный процессор Motorola 68000 (с операционной системой Unix и языками программирования Сн, Паскаль и Фортран-77), которые могут работать параллельно [Greeger, 1983].

Специальный блок системы LAMBDA обеспечивает интерфейс между шиной NuBus и шиной Multibus, которая организует доступ к периферийному оборудованию. Другой важный аспект разработки — интерфейс LAMBDA-системы с Ethernet II (через шину Multibus), которая, в свою очередь, обеспечивает протокол Arpanet. Таким образом, NuBus-архитектура позволяет подключить Лисп-процессор к существующим системам ЭВМ и сделать его доступным даже из сетевого окружения Arpanet. Не зависящая от устройств NuBus-архитектура разрабатывалась первоначально в МТИ, а в настоящее время поддерживается и развивается фирмой TI.

Собственно Лисп-процессор системы состоит из четырех плат, объединенных через отдельную шину. Процессор рассчитан прежде всего на интерпретацию команд, в которые Лисп-программа переводится компилятором, однако процессор может легко адаптироваться к созданию развитых архитектур для языков высокого уровня и оптимизироваться для специальных приложений путем микропрограммной поддержки. Проект представляет собой модульную расширяемую Лисп-машину с возможностями мультипроцессорирования благодаря комбинации LAMBDA-процессора с шиной NuBus.

Одна из самых производительных коммерческих Лисп-машин в мире ELIS создана фирмой Nippon Telegraph & Telephone Corp. (NTT) в Японии. Эта двухпроцессорная машина (рис. 6.9) включает 64-разрядный центральный процессор и периферийный препроцессор ввода-вывода, построенный на базе микропроцессора MC68010. Все внешние устройства и средства связи с локальной сетью подключены к периферийному процессору, а его шина соединяется с шиной центрального процессора через канал прямого доступа к памяти. Микропрограммный Лисп-процессор имеет память управления большой емкости, а также регистровый стек емкостью 32 Кслов (32-разрядных), построенные на высокоскоростных микросхемах статических запоминающих устройств (3V). В первом образце машины около половины центрального процессора реализовано на разрядно-модульных микропроцессорах семейства 2900 фирмы Advanced Micro Devices Inc. (AMD), а остальная часть — на схемах Шотки/ТТЛ. Таким образом, все логические схемы располагаются на одном кристалле, насчитывающем 20 тыс. вентиляей. Настольный компьютер ELIS может выполнять 1 млн. базовых Лисп-команд в секунду, т. е. примерно в 6 раз превосходить по скорости действия существующие Лисп-машины и мини-суперкомпьютеры [Osato et al., 1983; 1984].

Последние промышленные Лисп-машины стали внедряться в универсальную вычислительную среду ЭВМ, объединяя таким образом преимущества архитектуры, ориентированной на Лисп, с развитой высокопроизводительной архитектурой локальных и универсальных сетей ЭВМ. Новая тенденция заключается в интеграции специализированной Лисп-архитектуры в традиционное программное и языковое окружение. Кроме того, Лисп должен быть адаптирован в существующем компьютерном окружении, чтобы исключить необходимость резкого изменения традиционного программного обеспечения и стать эволюционирующим средством привнесения искусственного интеллекта в существующие компьютерные системы.

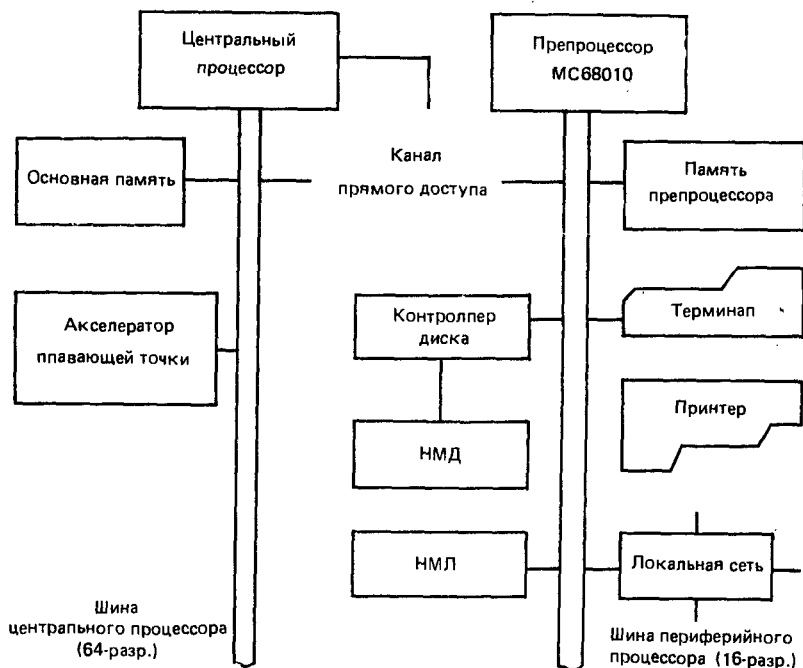


Рис. 6.9. Схема Лисп-машины ELIS фирмы NTT

**Процессоры, сочетающие числовую и символьную обработку.** Постепенно преодолевается представление потребителей о том, что Лисп не нужен при создании реальных прикладных систем, поскольку он недостаточно экономичен и надежен, не обладает удовлетворительными динамическими характеристиками и плохо совместен с традиционными языками [Электроника, 1986].

Поскольку Лисп является перспективным языком, предназначенным для интеллектуальных систем, одно из важных применений специализированных Лисп-процессоров в универсальных ЭВМ массового использования — поддержка средств интеллектуального интерфейса с пользователем.

В интеллектуальных системах все большее значение приобретает возможность совмещения обработки числовых и символьных данных. Символьные процессоры, по-видимому, будут все шире использоваться в качестве встроенных сопроцессоров. Тенденция к комбинированию средств обработки числовых и символьных данных прослеживается при необходимости оперативного анализа исходных данных и принятия управляющих решений.

Указанным требованиям удовлетворяет разработанная фирмой Integrated Inferegence Machines Inc. (IIM) 40-разрядная машина SM45000. Высокоскоростной и недорогой символьный процессор подключен к специализированному вычислительному процессору, обеспечивающему обработку числовых данных со скоростью 4 МФлопс. Кроме того, в структуру машины входит процессор 80286 фирмы Intel, управляющий вводом-выводом, в том числе файловой системой, совместимой с IBM PC. Процессор ввода-вывода и вычислительный процессор поддерживают традиционные языки числовой обработки. Символьный процессор SM45000 содержит оригинальные быстродействующие схемы для декодирования тегов, при более низкой цене на некоторых задачах до



8 раз превышает по производительности существующие символьные компьютеры, в частности модели 3600 фирмы Symbolics [Мануэль, 1987].

В этом направлении сейчас работают и другие компании, в том числе фирма TI. В 1986 г. фирма TI создала систему, объединив в единой вычислительной среде символьный Лисп-процессор Explorer-LX и сопроцессор на базе Motorola 68020 (с тактовой частотой 16 МГц), работающий с операционной системой Unix System V [Catier, 1987]. Эта двухпроцессорная система названа TI System V, в ней доступны традиционные языки программирования Фортран, Кобол, Паскаль, Сн. Система предназначена для тех приложений, в которых одновременно требуются программы представления знаний, написанные на Лиспе, и вычислительные программы на традиционных языках программирования.

Разработка инструментальных средств, обеспечивающих естественную связь между системами представления знаний и традиционными программными средствами, представляет главную проблему. Специалисты предполагают, что в 90-х годах фирмы DEC, IBM и другие, по-видимому, будут вводить процессоры для символьной обработки в состав архитектуры своих систем.

**Рабочие станции.** Характерным примером такого рода систем являются рабочие станции, ориентированные на решение задач автоматизации проектирования. Рабочие станции имеют развитые средства машинной графики и достаточно мощные Лисп-процессоры.

Одной из первых фирм, начавших выпуск рабочих станций на базе Лисп-процессоров, является Xerox Corporation, которая с 1981 г. выпускает рабочие станции серии 1100 (1108, 1100, 1132). Фирма Xerox ориентируется на выпуск рабочих станций для CAIP на базе как Interlisp, так и Common Lisp. Упомянутые модели имеют оперативную память емкостью 512 Кслов (16 разрядных) и содержат аппаратные средства, обеспечивающие управление виртуальной памятью. Ориентированная на Лисп система команд процессора для этих станций реализована микропрограммно. Фирма Xerox продолжает совершенствовать эту серию и недавно выпустила еще две модели: 1185 и 1186. Модель 1185 предназначена для исполнения готовых программ и является самой дешевой в серии 1100. Модель 1186 может использоваться также для проектирования интеллектуальных систем. Один из способов использования дешевых рабочих станций — включение их в комплект поставки программного продукта. Так, фирма Xerox заключила контракт на продажу 1000 своих рабочих станций 1185 с фирмой, которая будет выпускать программное обеспечение, создаваемое на языке Лисп, и продавать его в комплекте с рабочей станцией [Haber, 1986; Catier, 1987]. Еще одной рабочей станцией, ориентированной на использование языка Лисп, является серия 4400 фирмы Tektronix, традиционно специализировавшейся на выпуске электронно-лучевых трубок, осциллографов, анализаторов и дисплеев.

**Персональные Лисп-процессоры.** По данным анализа производительности некоторых современных универсальных ПЭВМ (например, IBM PC/AT, Macintosh фирмы Apple Computer, Tektronix 4404, SUN-3 фирмы Sun Microsystems, машины фирмы DEC и компьютеры на базе микропроцессоров Motorola 68010 и 68020), они лишь в 10 раз уступают по своей вычислительной мощности машинам серии Symbolics 3600 и вполне пригодны для реализации интеллектуальных систем [Мануэль, 1985; Электроника, 1986].

Использование ПЭВМ как базы для реализации Лиспа сейчас ограничивается недостатком ресурсов этих машин, что не позволяет иметь характерные для Лисп-машин развитые средства, ориентированные на разработку программного обеспечения. Специально ориентированная на Лисп-окружение ПЭВМ Explorer, разработанная фирмой TI, значительно менее мощная, чем машины фирм LMI и Symbolics, включает 32-разрядный микропрограммируемый Лисп-процессор, подсоединенный (как и в проекте LAMBDA) к шине NuBus, обеспечивающей выход в сеть ЭВМ [Bond, 1984; Matthews et al., 1986]. Поскольку полная реализация диалекта Zetalisp содержит в кодах приблизительно 4 млн. строк (больше, чем может обработать Explorer), то фирма TI выбрала в каче-

стве входного языка для своего изделия диалект Common Lisp, являющийся подмножеством Zetalisp, занимающий в кодах около 200 тыс. строк и программно совместимый с входным языком LAMBDA-процессора.

Учитывая появление наряду с большими мощными Лисп-системами промышленных образцов малых Лисп-машин, следует отметить, что малые Лисп-машины (типа Explorer) необходимы в качестве инструмента для отладки программ, проверки частных решений в учреждениях и лабораториях, для обучения специалистов и приобретения опыта работы с языком Лисп. Такие машины более дешевы и могут выпускаться в больших количествах. Например, Explorer позволяет пользователям работать с интерактивным дисплейным редактором, интерпретатором, компилятором и отладчиком. Его программное обеспечение включает средства форматирования текстов и машинную графику, интерфейс с естественными языками и возможности представления таблиц решений, обеспечение для языка Пролог (интерпретатор и средства для построения экспертных систем), средства описания семантических сетей. Хотя Лисп-машины уже достаточно распространены на рынке вычислительной техники, их численность по сравнению с 32-разрядными универсальными автоматизированными рабочими местами (АРМ) остается крайней малой.

Фирмы-производители новых универсальных 32-разрядных микропроцессоров пытаются конкурировать с производителями специализированных Лисп-микропроцессоров. Например, фирма Intel (совместно с другими) ведет разработку средств реализации языков Лисп и Пролог для микропроцессора 80386. Фирма надеется таким образом улучшить перспективы сбыта своей микросхемы как предназначенной для решения задач искусственного интеллекта и альтернативы Лисп-машинам. Для подтверждения основательности такого утверждения достаточно отметить, что на некоторых примерах машина Symbolics 3600 показала лишь в 2—2,5 раза более высокую производительность, чем процессор, построенный на базе Motorola 68020, работающий с тактовой частотой 16 МГц. В процессоре Motorola 68020 были введены быстродействующие команды выделения битов и перехода по группе разрядов, использовалась внешняя кэш-память емкостью 16 Кбайт и специализированное устройство управления памятью, обеспечивающее время обращения к ней 185 нс, а также небольшая внутренняя кэш-память [Мануэль, 1985; Диринг, 1987].

**Лисп-процессоры на базе RISC-архитектуры.** Еще одно направление Лисп-машин базируется на ставшей в последнее время популярной идее использования процессоров с сокращенной системой команд — RISC-процессоров. Принципы, лежащие в основе указанного подхода, состоят в следующем:

- в аппаратуре реализуется небольшой набор простых и регулярных команд, выполняемых за один такт;

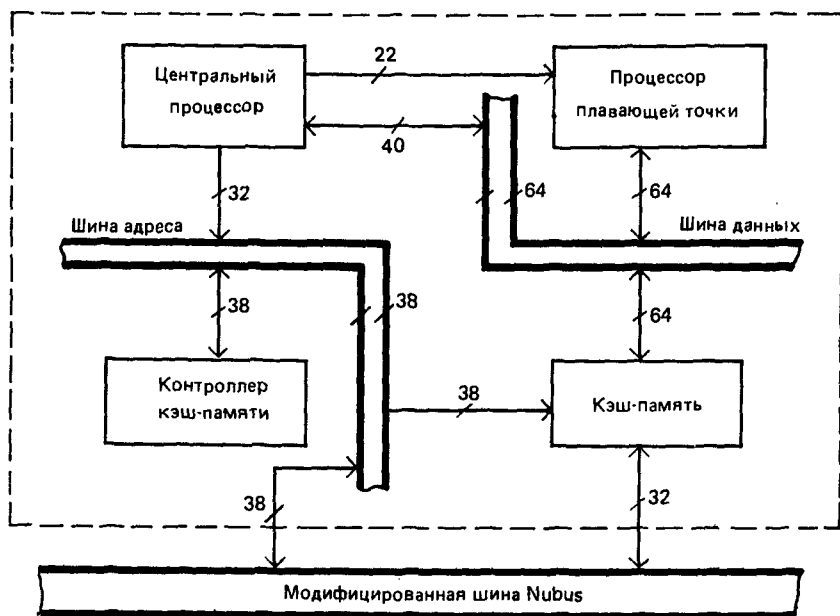
- в ЭВМ используется, как правило, аппаратное, а не микропрограммное управление;

- доступ к памяти имеет ограниченное число команд, другие команды работают только с регистрами;

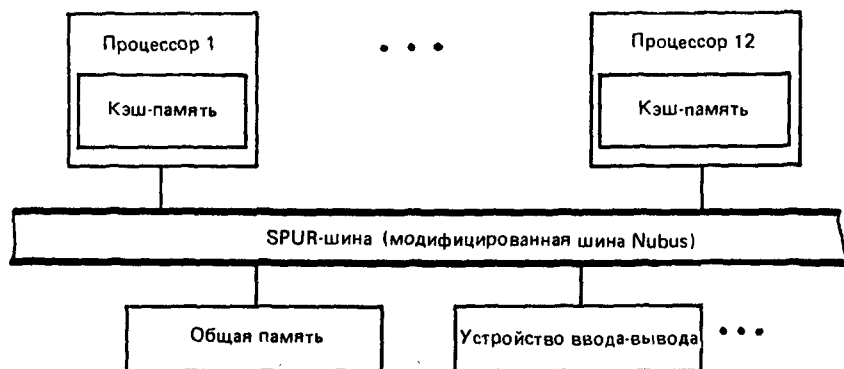
- команды ЭВМ удобны для эффективной компиляции языков высокого уровня.

Использование RISC-архитектуры при реализации Лисп-машин оправдано тем, что в основе Лиспа небольшое число сравнительно простых функций. Основной упор в таких реализациях Лиспа делается на компиляцию Лисп-программ. Для повышения эффективности к традиционным для RISC-процессоров командам добавляются команды, ориентированные на работу со списками и стеком [Bell, 1986].

Лисп-процессор (рис. 6.10,а), разработанный в Калифорнийском университете для рабочей станции SPUR (Symbolic Processing Using RISC), является процессором с сокращенной системой команд, специально приспособленной для реализации языков Лисп и Си. В систему входит до 12 процессорных элементов (рис. 6.10,б), работающих с 40-разрядным словом. Компилятор переводит Лисп-программу в команды процессора, среди которых есть команды, специально ориентированные на Лисп [Taylor et al., 1986; Hill et al., 1986].



а)



б)

Рис. 6.10. Схемы процессора SPUR-системы (а) и мультипроцессорной рабочей станции SPUR (б)

Следуя существующей тенденции использования RISC-архитектуры при проектировании интеллектуальных систем, подобную машину разрабатывают в университете Карнеги-Меллона [Catier, 1987]. RISC-архитектура позволяет разместить в кристалле ограниченного объема законченный Лисп-процессор и сделать реализацию всей машины более компактной.

**Лисп-процессоры на СБИС.** Одним из перспективных направлений в настоящее время считают однокристалльные специализированные Лисп-процессоры. СБИС, содержащая такой процессор, может встраиваться в разнообразные

изделия, что позволит создавать экономически выгодные интеллектуальные системы в самых разных областях применения.

Фирма TI разработала однокристалльный Лисп-микропроцессор для ЭВМ CLM (Compact Lisp Machine) на базе современной технологии (КМОП) с проектными нормами 2 мкм. Целью проекта являлось создание компьютеров размером с коробку для обуви и обеспечение возможности символьных вычислений в реальном масштабе времени. В 1986 г. фирма TI объявила о выпуске этого первого в мире однокристалльного Лисп-микропроцессора с внутренней оперативной памятью 2 Мбайт, работающего на частоте 40 МГц. Данная СБИС приблизительно в 10 раз превосходит по сложности микропроцессор Motorola 68000 и является ключевым компонентом ЭВМ CLM (ее развитием будет ARIES — встроенная интеллектуальная система), представляющей собой комплект из четырех плат размером 167×149 мм каждая [Tucker, 1986; Электроника, 1986, 1987]. Эта 32-разрядная СБИС, включающая 550 тыс. транзисторов, является несколько упрощенным вариантом двухплатного центрального процессора, применяемого в АРМ Explorer. На кристалле Лисп-процессора размещается более 60% схем центрального процессора АРМ Explorer, однако он приблизительно в 5 раз превосходит АВМ Explorer по производительности. Кроме Лисп-процессора в комплект входят следующие функциональные модули: модуль памяти с блоком управления оперативной и кэш-памятью, интерфейс с шиной, модуль технического обслуживания и диагностики и источник питания с системным коммутатором. Полная реализация указанного проекта позволит создавать экспертные системы, которые можно разместить в ограниченном пространстве самолетов и танков. Хотя процессор ориентирован на военные цели, он должен открыть много новых приложений.

Новый Лисп-микропроцессор был встроен в АРМ Explorer, что позволило создать более производительный процессор Explorer-II, появление которого на рынке способствовало повышению активности в области разработки прикладных интеллектуальных систем. Все программные средства АРМ Explorer могут функционировать на новом Лисп-микропроцессоре. Применение фирмой TI КМОП-технологии с двухслойной металлизацией и проектными нормами 1,2 мкм позволит разместить на кремниевом кристалле Лисп-процессора площадью 1 см<sup>2</sup> более 550 тыс. транзисторов. Это будет, видимо, один из самых насыщенных промышленных микропроцессоров в мире. Длительность такта такого микропроцессора составит 40 нс (а в дальнейшем будет сокращена до 30 нс).

Процессорный модуль, содержащий четырехуровневый конвейер, позволит выполнять одну микрокоманду за один такт, а блок очереди упреждающей выборки макрокоманд позволит за один такт выполнить наиболее часто используемые небольшие макрокоманды, состоящие из нескольких микрокоманд. На кристалле центрального процессора размещается часть памяти емкостью 114 Кслов, выполняющая функции кэш-памяти. Основным исполнительным элементом устройства является АЛУ с 32-разрядной регистровой циклической схемой сдвигатель/маскирование. Модуль основной памяти системы ARIES имеет емкость 8 Мбайт (с возможностью расширения до 64 Мбайт). Внутренние обмены данными между модулями системы осуществляются по 32-разрядной шине NuBus. Первые промышленные образцы системы ARIES должны появиться в 1989 г., их стоимость по предварительным оценкам составит около 100 тыс. дол. Система ARIES будет использоваться для экспертных систем, диагностирующих и обслуживающих бортовое оборудование самолетов и другое электронное военное оборудование.

Основной конкурент TI — фирма Symbolics — также стремится внедрить в свои последние машины серии 3600 СБИС, чтобы уменьшить размеры, стоимость и повысить производительность Лисп-процессора, совместимого с существующими моделями серии 3600.

Машины на базе СБИС позволят перейти от больших дорогих систем (ценой от 30 до 100 тыс. дол.) к менее крупным и недорогим. Специалисты считают, что будущее увеличение темпов роста интеллектуальных систем связано с появлением более компактных интеллектуальных машин.

**Параллельные Лисп-процессоры.** Поскольку дальнейшее повышение быстродействия процессоров последовательного действия сопряжено с серьезными технологическими трудностями, получают распространение параллельные реализации ЭВМ, в том числе Лисп-процессоров. По мнению ряда специалистов, параллелизм остается, видимо, единственным способом повышения эффективности символьных вычислений. Однако по некоторым оценкам для большинства программ, написанных на традиционных языках программирования, использование параллельных процессоров обеспечивает не более чем 4-кратное повышение быстродействия [Диринг, 1987].

Трудности реализации параллельных символьных вычислений связаны с семантикой языков, со стилем традиционного программирования, алгоритмами, не приспособленными для работы в параллельном режиме, использованием архитектур ЭВМ, непригодных для параллельной обработки (в частности, архитектуры с общей памятью). Обмены данными между многочисленными процессорами параллельной системы должны происходить по специальным каналам связи. Трансляторы с языков искусственного интеллекта должны обеспечивать деление программ на сегменты необходимых размеров.

К параллельному программированию с применением интеллектуальных систем, в частности к разработке параллельной версии языка Лисп и адаптации его к архитектурам параллельных машин, проявляет интерес ряд фирм. Средства для параллельной обработки реализованы, например, в языке Multilisp, разработанным в МТИ. Японские специалисты называют машинами пятого поколения именно параллельные компьютеры, оснащенные средствами искусственного интеллекта, над созданием которых они активно работают.

Первым коммерческим применением параллельного Лиспа в многопроцессорной системе является разработанная и предложенная пользователям в 1986 г. система параллельных компьютеров iPSC фирмы Intel, построенных с использованием матрицы процессоров 80286 [Электроника, 1986].

Введение каналов связи между этими процессорами позволило фирме Gold Hill Computers Inc. (совместно с Intel) реализовать параллельную версию языка Common Lisp. Разработчики взяли за основу язык Common Lisp для одноканальных процессоров 80286, который используется в ПЭВМ, и ввел такую дополнительную возможность, как обмен информацией между узлами, осуществляемый по широкополосному каналу. Созданные программные средства могут распределять задачу между многими процессорами. Фирма Intel совершенствует свой компьютер iPSC, повышая быстродействие, наращивая память (до 4 Мбайт на каждый элемент). Современный 128-процессорный вычислительный комплекс имеет производительность до 100 млн. опер./с.

В мире получают распространение проекты, направленные на создание символьных процессоров высокого класса, которые более чем в 100 раз превосходили бы ныне существующие (например, VAX-11/780 фирмы DEC) и были бы пригодны для будущих приложений. Они, как правило, представляют собой набор одинаковых процессоров, объединенных в сеть. Каждый из них может, подобно отдельному компьютеру, выполнять в последовательном режиме оттранслированную программу, хранящуюся в его памяти. Отдельные процессоры взаимодействуют друг с другом посредством сообщений, посылаемых через порты ввода-вывода. При этом каждый элемент, в свою очередь, может представлять собой несколько параллельно работающих подсистем, осуществляющих определенные процедуры символьной обработки.

#### **Структурные и функциональные особенности Лисп-процессоров**

Последовательный анализ языка и ряда реализаций позволяет выделить основные архитектурные элементы Лисп-процессора, отражающие наиболее существенные функции и свойства языка.

Архитектура Лисп-процессора может быть представлена как совокупность специализированных модулей, каждый из которых предназначен для выполнения определенной функции (рис. 6.11):

препроцессирование, или процессор трансформации (ПТ) исходного текста программы;

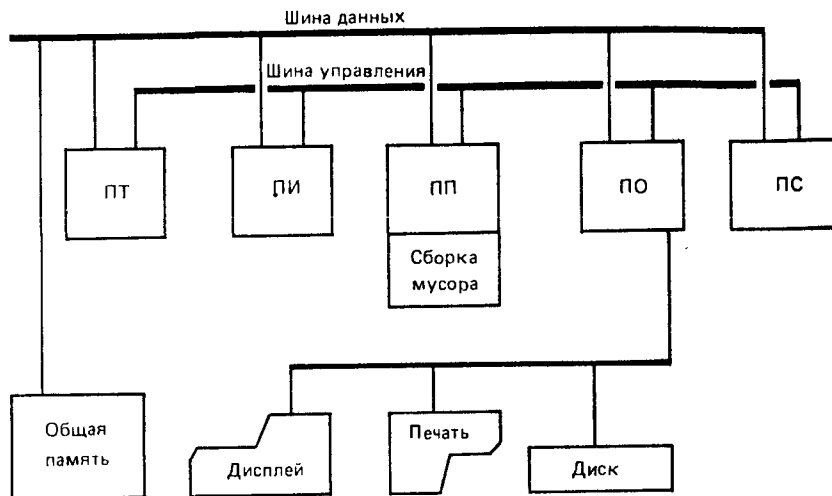


Рис. 6.11. Обобщенная схема Лисп-системы

исполнение, или процессор интерпретации (ПИ);  
управление памятью, или процессор памяти (ПП), включающий сборку мусора;

ввод-вывода, или процессор обмена (ПО) или интерфейсный процессор; техническое обслуживание и диагностика, или сервисный процессор (ПС).

Проведение функциональных границ между модулями осуществляется так, чтобы минимизировать потоки данных между ними и упростить управление. Взаимодействие модулей осуществляется через одну или несколько шин, обеспечивающих архитектурную гибкость. Подобная комбинация компонентов обычно допускает параллельное осуществление интерпретации, сборки мусора и ввода-вывода. В конкретных проектах Лисп-процессоров могут присутствовать или все, или некоторые из указанных модулей, а другие при этом могут быть конструктивно объединены вместе; модули могут быть реализованы на одном и том же либо различных физических процессорах.

Можно ожидать, что дальнейшее развитие такого подхода приведет к разработке функционально-ориентированных БИС, используя которые можно будет создать полную Лисп-систему, объединяя эти БИС на общей шине.

Специалисты отмечают, что Лисп-системы на базе СБИС в перспективе могут революционизировать развитие и внедрение интеллектуальных средств (как это сделали ПЭВМ в отношении средств автоматизации проектирования). Они позволят оснастить реально доступными новыми средствами автоматизации рабочее место инженера.

В перспективе предусматривается не только снижение цен на Лисп-процессоры (примерно на 35% в год, гораздо быстрее, чем для обычных ЭВМ), но и сопряжение средств символьной обработки с более традиционными программными средствами числовой обработки. Новая тенденция заключается в интеграции специализированной Лисп-архитектуры в традиционное программное и языковое окружение. В течение последних лет рынок Лисп-процессоров увеличивался ежегодно на 50% и в настоящее время оценивается в 120 млн. дол. Согласно прогнозам специалистов, через 10—15 лет символьные процессоры составят около половины всех вычислительных машин.

## 6.2. Пролог-машины и спецпроцессоры вывода

*В. Н. Вагин, В. Н. Захаров*

Пролог — язык высокого уровня, основанный на математической логике. Главной его особенностью является наличие нескольких равноправных семантик для его текстов. Текст на Прологе может трактоваться как декларативное описание некоторых закономерностей и как описание некоторых процедур [Colme-gauer et al., 1981].

Спецпроцессоры, использующие язык Пролог, или Пролог-машины, ориентируются на решение задач, связанных с логическим выводом, что позволяет надеяться на решение задач дедукции практической сложности (правда, для ограниченного языка исчисления предикатов первого порядка — хорновских дизъюнктов).

Работы по созданию Пролог-машин начались в рамках известного японского проекта по разработке вычислительных систем пятого поколения. Вычислительные системы пятого поколения будут ориентированы на обработку знаний и предполагать весьма развитыми возможностями логического вывода. Исследования

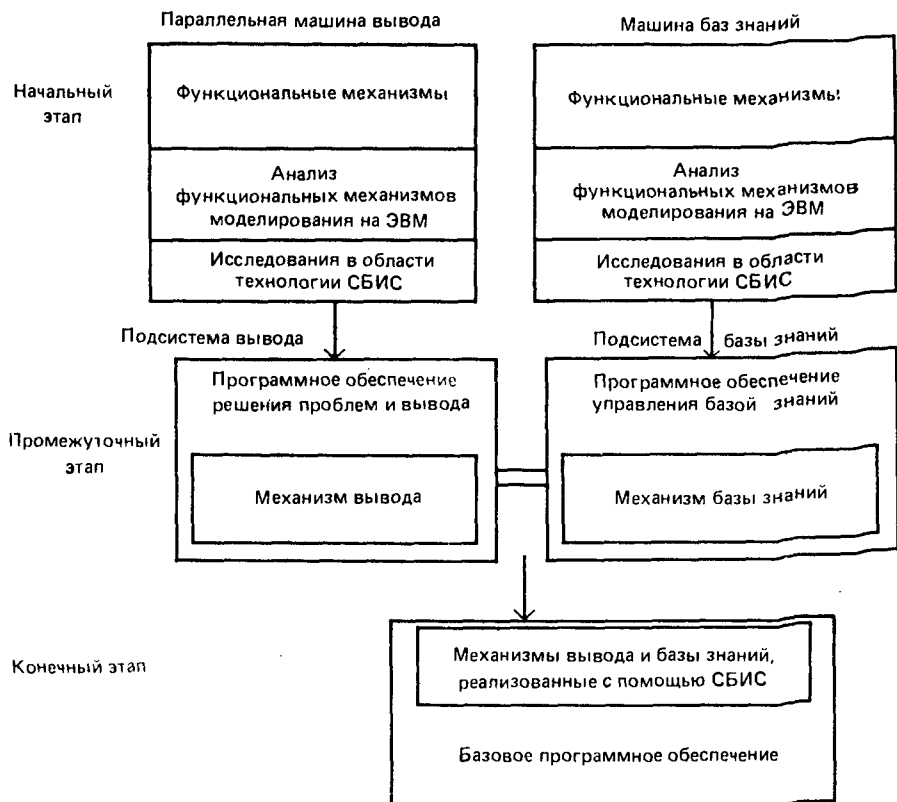


Рис. 6.12. Этапы разработок Пролог-машин

по этой проблеме планируются на десятилетний срок начиная с 1982 г., и включают три этапа (рис. 6.12) [Mugakami et al., 1985].

В начальный период (1982—1985 гг.) были исследованы основные компоненты параллельной машины вывода и машины баз знаний. Каждую из этих подсистем планируется построить в промежуточный четырехлетний период и затем интегрировать их в единую систему к 1992 г. Таким образом, основной задачей этого проекта является создание высокопроизводительной параллельной персональной Пролог-машины и машины баз знаний. Хотя официально ответствен за проект лишь институт ICOT (Япония), большинство исследований ведется в других фирмах, среди которых такие промышленные гиганты, как Fujitsu, Mitsubishi, NEC и др. Это говорит о том, что многие результаты исследований остаются секретами фирм. Таким образом, проект машины 5-го поколения имеет несколько этапов, одни из которых открыты, а другие — закрыты для исследователей.

### Существующее положение и реализации

Одним из наиболее известных результатов является последовательная персональная машина вывода PSI, разработанная фирмой Mitsubishi в 1983 г. [Mugakami et al., 1984; 1985], которая работает в комбинации с машиной реляционной базы данных DELTA (рис. 6.13). В данной комбинации машина PSI является хост-машиной.

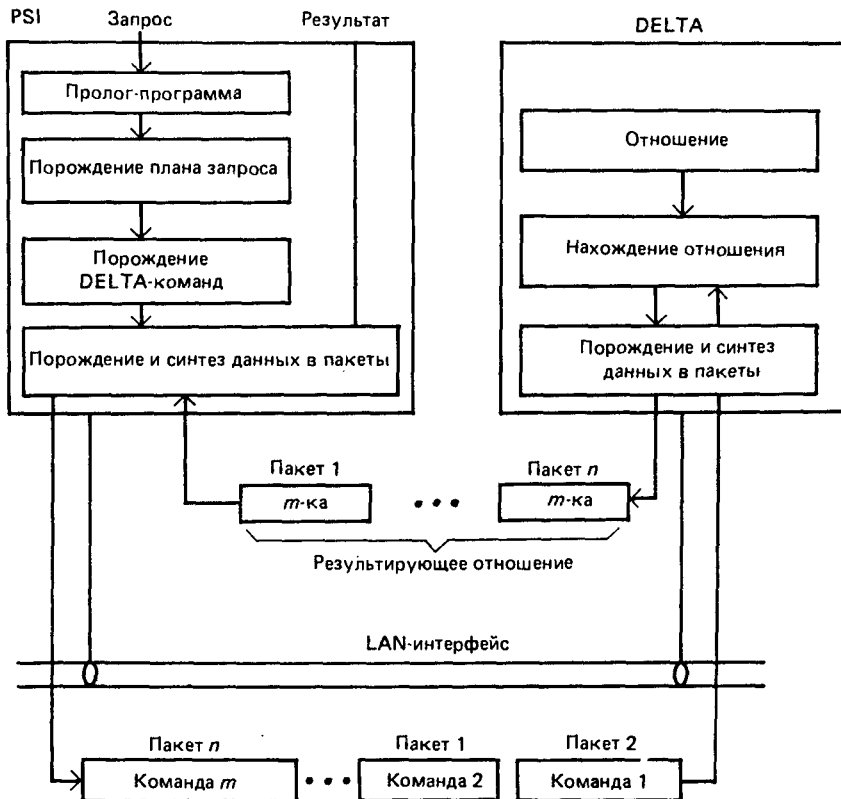


Рис. 6.13. Схема машины вывода PSI



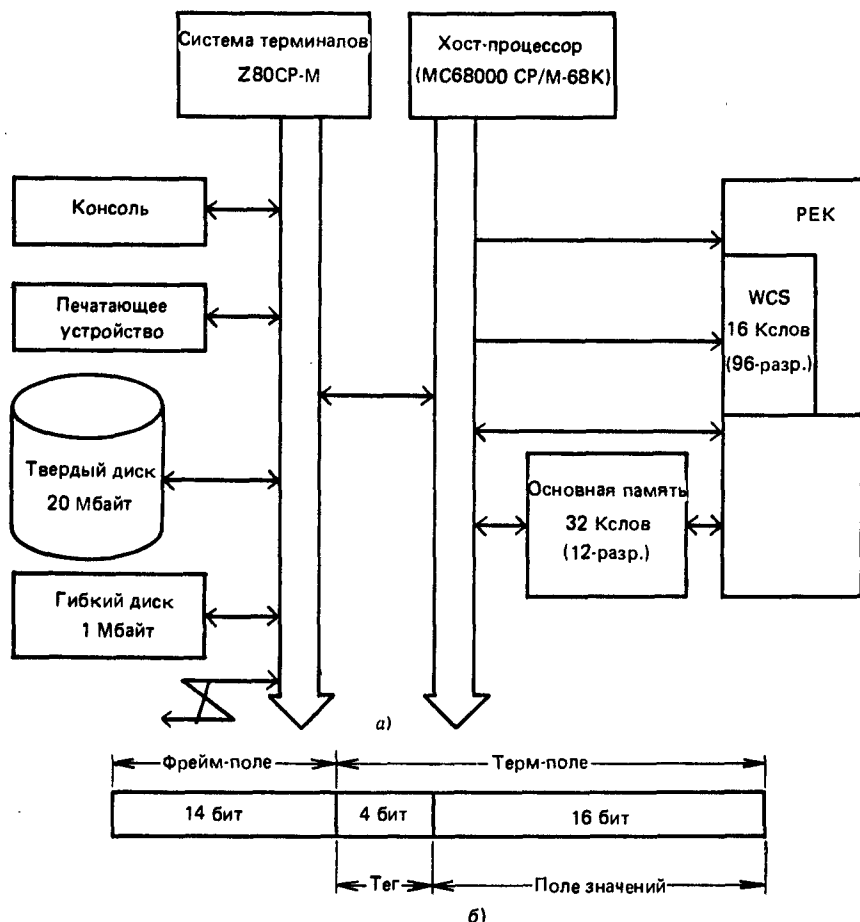


Рис. 6.14. Схема (а) и формат слова (б) машины вывода РЕК

Поток заявок идет от хост-машины к DELTA. Пользователь хост-машины пишет свою программу на языке Пролог и делает запрос к внешней базе данных. Программное обеспечение машины PSI порождает планы запроса согласно его содержанию, затем преобразует их в команды машины DELTA в виде пакетов и осуществляет запрос к DELTA через LAN-интерфейс. Машина DELTA, в свою очередь, извлекает эти команды из пакетов и после анализа находит нужные данные в базе данных, передавая их затем хост-машине через LAN-интерфейс. Как считают авторы проекта, такая комбинация машин позволит более полно исследовать проблему обработки знаний и может стать экспериментальной средой для создания более производительных машин обработки знаний.

Другим примером последовательной Пролог-машины является машина РЕК, разработанная в Университете Кобе (Япония) (рис. 6.14,а) [Kaneda et al., 1986]. Хост-процессор управляет РЕК-процессором, который является процессором заднего плана для хост-процессора. В качестве хост-процессора, работаю-

шего с операционной системой CP/M-68K, используется микропроцессор MC68000.

В аппаратуре машины РЕК применяются микрокоманды горизонтального типа; распределенная память; теговая архитектура; аппаратные схемы для унификации и механизма возврата.

Программное обеспечение машины РЕК состоит из программной поддержки для отладки системы и Пролог-интерпретатора, выполняющего программы на языке Пролог с быстродействием 60—70 КЛипс, за 89 микрокоманд один логический вывод.

Микропрограммное устройство последовательного действия (sequencer) РЕК-процессора использует четыре квантованных по разрядам устройства типа Am2909A и генерирует 14-разрядные микропрограммные адреса.

Микропрограммная память РЕК-процессора является памятью управления записью (WCS) из 16 Кслов. Загрузка микропрограмм в память WCS осуществляется под управлением хост-процессора. Микрокоманды длиной 96 разрядов, разделенных на 24 поля, могут запускать параллельно ряд схем. Синхрогенератор (Am2925) генерирует четырехфазные синхросигналы. Время цикла  $120 + i \cdot 40$  нс ( $i=0, 1, \dots, 7$ ) и определяется микрокомандой. Выполнение микрокоманды и чтение следующей микрокоманды перекрываются (одноуровневый конвейер). Микрокоманда использует один из следующих адресов перехода:

содержимое микростека микропрограммного устройства последовательного действия;

содержимое регистров АЛУ, памяти процессов или стека (данные S-шины); непосредственные данные поля в микрокоманде.

Условия перехода включают 36 типов (используется Am2904). Четырехразрядный выход схемы сопоставления соединяется с входным терминалом Am2909.

Слово РЕК-процессора (рис. 6.14,б) состоит из 14-битового фрейм-поля и 20-битового терм-поля. Терм-поле, в свою очередь, включает 4-битовую теговую часть и 16-битовое поле значений. Тег указывает тип данных, значения имеют различный смысл в зависимости от тега. Терм-поле служит для хранения переменной или некоторой структуры, зависящей от переменных. Фрейм-поле указывает адрес фрейма переменной.

Память РЕК-процессора состоит из ряда модулей:

WCS — служит для хранения микропрограмм; хост-процессор может как считывать из WCS, так и записывать в WCS;

общая память — для хранения структуры данных, головных частей атомов и Пролог-программы;

память процессов — для хранения управляющей информации, полученной при выполнении Пролог-программ;

глобальный стек (34 разр.×16 Кслов) для хранения глобальных и локальных переменных;

хвостовой стек (14 разр.×16 Кслов) — для хранения адресов переменных;

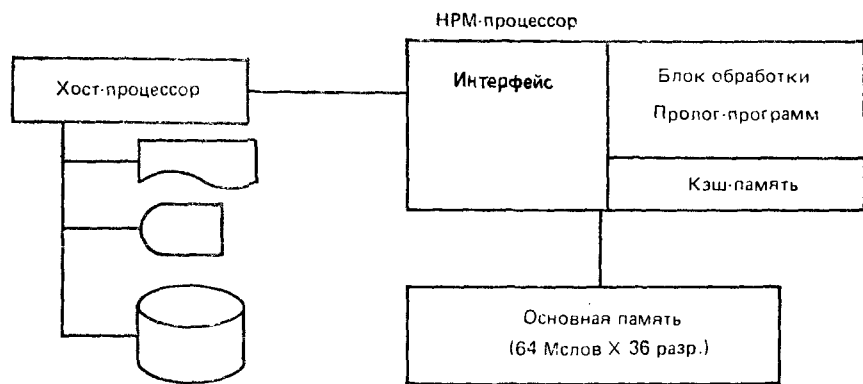
аппаратурный стек (34 разр.×4 Кслов) — для унификации;

файл регистров (34 разр.×32 Кслов).

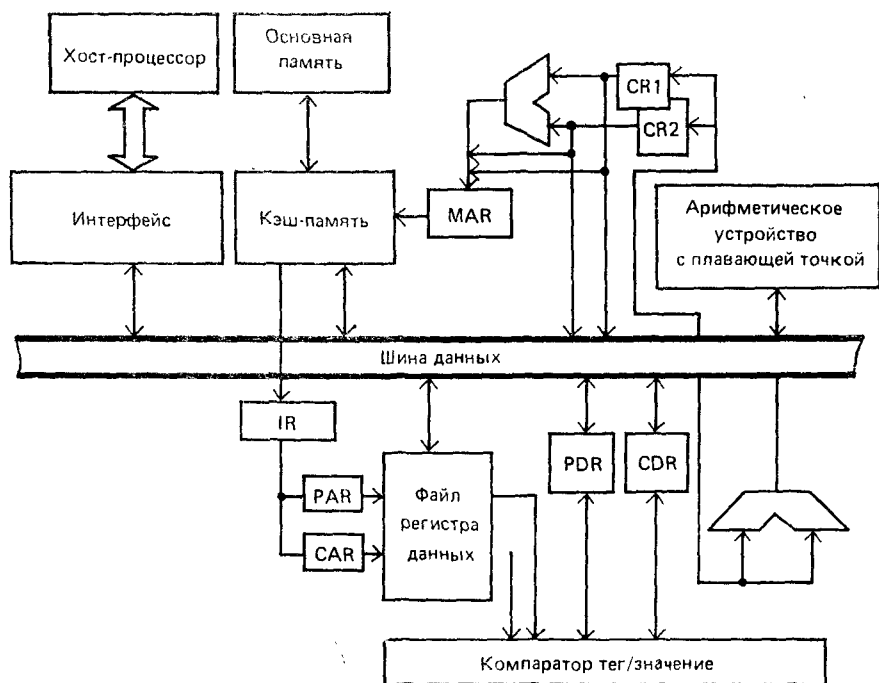
Возможен параллельный доступ к двум или более памятям. Для выполнения высокоскоростной унификации РЕК-машина снабжена схемами для конвейерного чтения структурных данных, схемами сопоставления, схемой автоматического вычисления адреса переменной.

Рассмотренные Пролог-машины предназначены для экспериментальных исследований и имеют довольно невысокое быстродействие и небольшую емкость рабочей памяти. Для реализации Пролог-программ, написанных для систем искусственного интеллекта, требуются машины, имеющие намного большее быстродействие и емкость рабочей памяти.

Известны по крайней мере две такие машины. Первая НРМ также развивается в рамках японского проекта по созданию машин пятого поколения [Nakazaki et al., 1985]. Главная ее особенность — память большой емкости и специализированная схема для выполнения операций унификации. Архитектура НРМ ориентирована на компиляторы с командами стекового управления высокого



а)



б)

Рис. 6.15. Схемы машины (а) и процессора (б) НРМ:

MAR — регистр адреса памяти; CR — регистры управления; IR — регистр команд; PAR — регистр адресов «родителей»; CAR — регистр текущих адресов; PDR — регистр данных «родителей»; CDR — регистр текущих данных

уровня. НРМ-процессор реализован на быстродействующих чипах типа CML (Current Mode Logic), имеющих машинный цикл 100 нс. Ее быстродействие оценивается в 280 КЛипс при выполнении детерминированной «сцепленной» Пролог-программы.

НРМ-процессор используется вместе с последовательной ПЭВМ PSI. Если PSI применяется для выполнения относительно малых Пролог-программ, то НРМ-процессор предназначен для решения больших Пролог-программ, которые невозможно решить на PSI. НРМ является процессором заднего плана, соединенным с PSI. PSI и НРМ отличаются уровнем машинных команд, причем уровень команд НРМ ниже, чем у PSI. Улучшение характеристик НРМ достигается главным образом благодаря применению методов оптимизации компилятора и специализированных аппаратных модулей. Хотя множество команд НРМ отлично от команд PSI, большинство Пролог-программ могут быть решены на обеих машинах (если возможности машины PSI позволяют это сделать). На рис. 6.15,а дана конфигурация НРМ-системы, в которой НРМ-процессор служит для выполнения Пролог-программ большого объема, а PSI, являющаяся хост-процессором, управляет операциями ввода-вывода.

НРМ-процессор (рис. 6.15,б) состоит из блока обработки Пролог-программ, кэш-памяти и интерфейса с хост-процессором. НРМ-процессор непосредственно интерпретирует данные внутренних объектов и коды, используя микропрограммы и специальную аппаратуру. Система памяти состоит из двух блоков основной памяти, блока управления системой и сервисного процессора. Внутренний код объектов и данные хранятся в блоках основной памяти, имеющей 36-разрядное слово.

Блок обработки Пролог-программ (Пролог-процессор) выполняет команды последовательно конвейерным способом за три стадии: выборка (ввод) команд, декодирование и выполнение. На стадии выборки команд регистр команд IR получает из кэш-памяти команду, состоящую из одного слова, которая затем декодируется и передается в регистры адресов «родителей» PAR и текущих адресов CAR. На стадии выполнения команды модули сопоставления аргументов и манипулирования стеками указателем применяются в основном параллельно. НРМ-архитектура требует довольно много регистров управления. Так, 48 регистров управления предназначены в качестве стековых указателей, рабочих регистров, используемых микропрограммой, и для других целей, определенных в НРМ-архитектуре.

НРМ-команды реализуют функции сопоставления аргументов, управления стеками и механизмом возврата. Емкость основной памяти НРМ, реализованной на чипах типа DRAM емкостью 256 Кбит, 64 Мслова (36-разрядных).

Кэш-память, содержащая команды и данные, имеет время доступа 100 нс, ее емкость 8 Кслов.

Другим примером быстродействующей Пролог-машины является процессор специального назначения PLM, предназначенный для высокопроизводительного выполнения Пролог-программ [Dobry et al., 1985]. Это первый прототип логического процессора в системе Aquarius — высокопроизводительной гетерогенной мультипроцессорной машины MIMD, разработанной в Беркли [Despain et al., 1984]. Цель проекта Aquarius — понимание принципов, положенных в основу функционирования машины, предназначенной для параллельного выполнения как символьных, так и числовых операций. Этот проект поддерживает принципы логического программирования на уровне управления и операции с массивами на функциональном уровне.

Система Aquarius состоит из хост-системы NCR/32, интерфейса PMI и Пролог-процессора PLM (рис. 6.16).

Блок PMI (PLM Memory Interface Unit) отвечает за все интерфейсные операции между Пролог-процессором PLM и РМ-шиной, ведает протоколами, организует буферный доступ к памяти пространства кодов и данных, а также реализует логику приоритетности и порядка следования к этим памяти. Необходимость буферизации при доступе к пространству данных возникает из разницы времен циклов между Пролог-процессором (100 нс) и хост-системой NCR/32

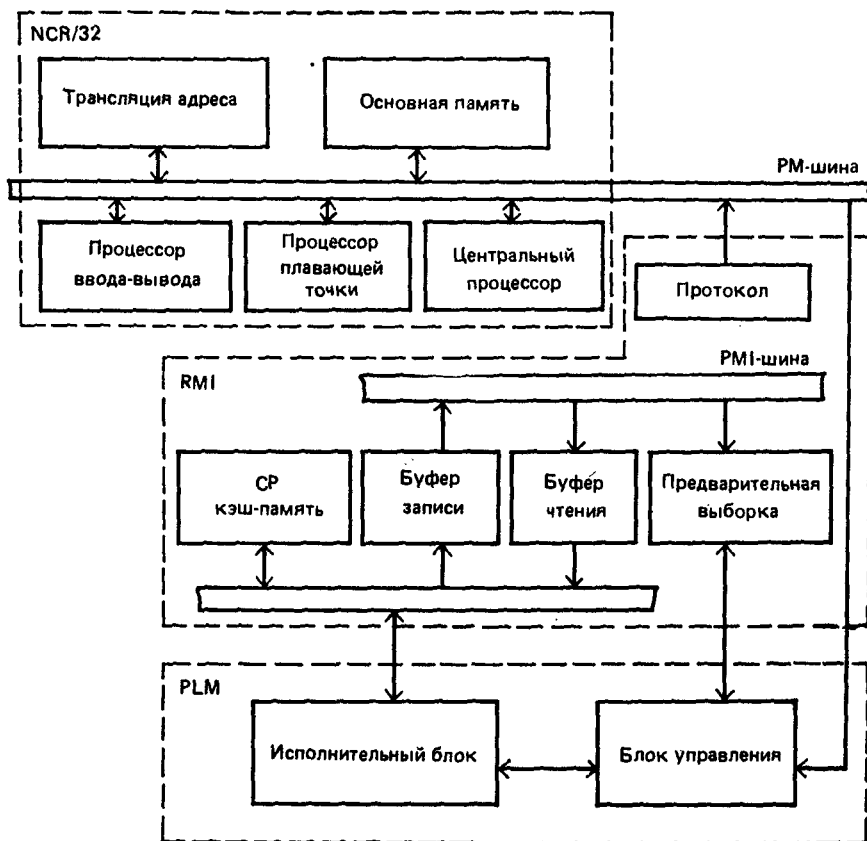


Рис. 6.16. Архитектура системы Aquarius

(150 нс). В кэш-память СР (Choice Point) записывается и хранится та подцель, с которой осуществляется альтернативный выбор (точка выбора) при выполнении Пролог-программы. Так как команды PLM имеют переменную длину, то для их выравнивания предназначен блок предварительной выборки, что позволяет использовать эти команды в блоке управления PLM в одном и том же формате.

Пролог-процессор PLM состоит из исполнительного блока и блока управления. Блок управления, в свою очередь, состоит из памяти управления, служащей для хранения горизонтального микрокода, и устройства для последовательного выполнения этого микрокода. По командам блока управления начинает функционировать исполнительный блок, который состоит из файла регистров и арифметическо-логического устройства (АЛУ). АЛУ выполняет арифметические операции и операции сравнения. Файл регистров содержит все регистры процессора, поддерживающие множество команд PLM, которые условно можно разделить на 6 групп: управление процедурами (Procedure Control), индексирование (Indexing), управление дизъюнктами (Clause Control), получение (Get), помещение (Put) и унификация (Unify).

Архитектура PLM была промоделирована на уровне множества команд и микрокоманд. Моделирующие программы написаны на языке C и работают на машине VAX-11/750 (Berkeley (4.2 BSD) Unix). Сравнив производительности машины PLM и других известных Пролог-машин (кроме HPM-процессора), авторы системы пришли к выводу, что компиляция множества команд языка Пролог позволяет повысить производительность в 20 раз по сравнению с интерпретирующим вариантом. Производительность PLM-процессора в среднем в 10 раз выше производительности машины PSI — 300 КЛипс вместо 30 КЛипс.

### Перспективы

Несмотря на большое число исследований, посвященных Пролог-машинам и спецпроцессорам вывода, проблемы построения высокопроизводительных машин вывода остаются в центре внимания многих исследователей. К наиболее важным проблемам, которые должны быть решены в ближайшее время, относятся задачи максимального распараллеливания программ, написанных на языке Пролог, создания более мощных, чем язык Пролог, языков параллельного логического программирования, разработки принципиально новых архитектур параллельных машин вывода.

Рассмотрим некоторые абстрактные модели параллельных машин вывода, которые могут быть реализованы в соответствующие высокопараллельные Пролог-машины. Остановимся прежде всего на архитектуре параллельной машины вывода, основанной на модели продукции целей [Goto et al., 1984]. Здесь используется метод доказательства для Пролог-программ, основанный на фреймовом представлении логических утверждений. Каждому хориовскому дизъюнкту сопоставляется фрейм, два типа вершин которого (полукруглости и маленькие окружности) соответствуют литерам и аргументам этих литер. Ребра связывают аргументы с их значениями или с их именами в дизъюнкте. Операция унификации является операцией сопоставления между аргументами соответствующих литер, что во фреймовом представлении означает соединение одной половинки окружности в целевом фрейме с другой половиной одного из исходных дизъюнктов, подходящего для такого соединения, и проверку на непротиворечивость подстановок. Имеет место следующие типы параллелизма:

между аргументами двух литер, сопоставляемых параллельно;

между шаблонами исходных дизъюнктов — некоторой целевой литере сопоставляется одновременно несколько кандидатов-дизъюнктов, чьи шаблоны совпадают;

между целевыми литерами — если в целевом фрейме имеется несколько целевых литер, то операция унификации для каждой целевой литеры может выполняться параллельно;

между целевыми фреймами — если имеется несколько целевых фреймов, то каждый из них может разрешаться параллельно.

Как правило, параллелизм первого типа не используется вследствие необходимости проверки на непротиворечивость при выполнении операции сопоставления аргументов двух литер. Не используется и параллелизм между целевыми литерами (AND-параллелизм) из-за больших трудностей, возникающих при параллельной обработке нескольких целевых литер, принадлежащих одному целевому фрейму. Главное внимание в этой абстрактной схеме машины вывода уделяется параллелизму между целевыми фреймами (OR-параллелизм).

Одним из основных элементов Пролог-машины PIE является унифицирующий процессор, выполняющий следующие процессы:

инициализацию — вызов целевого фрейма, отделение одной литеры от другой и вызов процесса сопоставления;

сопоставление целевого фрейма с одним из кандидатов исходного множества дизъюнктов, в случае успешного сопоставления вызывается процесс редукции;

редукцию — генерирование нового целевого фрейма из исходного целевого и дизъюнкта-кандидата.

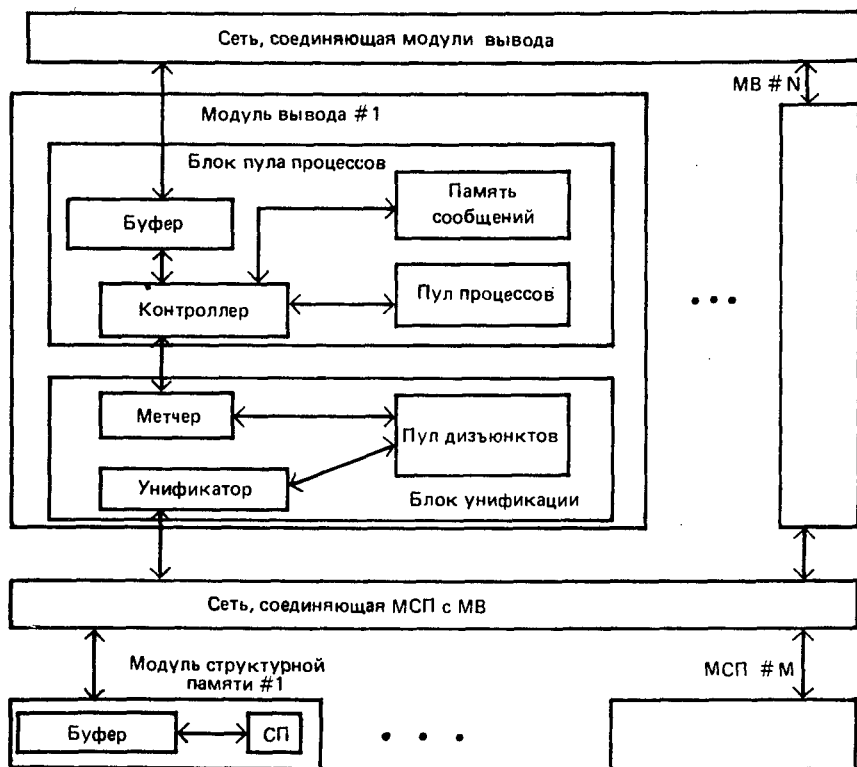


Рис. 6.17. Архитектура машины PIM-R

Унифицирующий процессор выполняет многократно операции унификации между входным целевым фреймом и всевозможными дизъюнктами-кандидатами, которые хранятся в памяти дизъюнктов, и при успешной унификации генерирует новые целевые фреймы. Полный механизм параллельной обработки целевых фреймов состоит из множества унифицирующих процессоров и пула целей, служащего для хранения множества целевых фреймов. Каждый унифицирующий процессор многократно вызывает из пула целей целевые фреймы и после редукции возвращает в него новые целевые фреймы.

Архитектура машины PИЕ поддерживает два главных требования, выдвигаемых при создании высокопараллельных машин вывода, — это локальность и независимость. Кроме того, такая архитектура делает ненужным механизм возвратов и порядок разрешения литер, но вызывает проблемы организации трафика в сети сообщений.

Другой абстрактной моделью параллельной машины вывода, где главное внимание также уделяется OR-параллелизму, является машина PИМ-R [Opai et al., 1985], которая состоит из модулей вывода и структурной памяти (МВ и МСП), а также сети, соединяющей эти модули (рис. 6.17). В свою очередь, модуль вывода состоит из блока унификации и блока пула процессов.

В блоке унификации пул дизъюнктов состоит из блока управления группой дизъюнктов и памяти, в которой хранятся сами дизъюнкты. Блок управления группой дизъюнктов служит для хранения числа дизъюнктов, находящихся

в OR-отношении указателя на память дизъюнктов, и типа данных первого аргумента головной литеры дизъюнкта. Блок унификации содержит также метчер (matcher) и унификатор. Метчер выбирает унифицируемые дизъюнкты согласно типу данных первого аргумента цели из блока пула процессов, посылает унификатору целевой дизъюнкт и позицию, где хранятся дизъюнкты-кандидаты. Унификатор унифицирует эти дизъюнкты и посылает результаты в выходной буфер.

Блок пула процессов состоит из пула процессов и памяти сообщений, а также контроллера процессов. Пул процессов является памятью для хранения процессов, включающих управляющую информацию для последовательности целевых дизъюнктов, шаблона последовательности целевых дизъюнктов и т. д. Контроллер процессов выполняет следующие функции: создание, возобновление и стирание процесса; реализацию локальных программно-исполняемых стратегий, использующих редукцию; сборку мусора. Каждый модуль вывода имеет распределенную память сообщений, служащую для хранения переменных специального типа.

Одним из важных требований, предъявляемых к архитектуре параллельных машин вывода, является требование эффективной обработки сложных и больших структурированных данных. Здесь возникают проблемы, связанные с копированием таких данных. Чтобы уменьшить затраты на копирование, вводится модуль структурной памяти, которая предназначена для хранения таких сложных структурированных данных, как большие комбинированные списки и векторы. Каждый модуль структурной памяти соединен с несколькими модулями вывода через соответствующую сеть. Все модули вывода соединены в клетчатую структуру, позволяя вершинам такой сети динамически управлять распределением процессов на каждом модуле вывода.

Для уменьшения затрат на копирование информации и для минимизации числа потоков сообщений, проходящих через сеть, в машине PIM-R используется копирование только редуцируемой цели в отличие от копирования всех целей в машине PIE.

Разновидностью архитектуры абстрактной машины PIM-R является архитектура PIM-D, в основу которой положена модель потоков данных (более подробно см. [Miyakami et al., 1985]).

Таким образом, создание новых высокопараллельных Пролог-машин позволит в будущем повысить производительность таких машин до  $10^8$ — $10^9$  Лисп, что эквивалентно производительности универсальных ЭВМ со скоростью работы до 100 млрд. опер./с. Ясно, что без специализации аппаратного обеспечения достичь такой производительности будет очень трудно.

Кроме того, язык Пролог как базовый язык для машин пятого поколения должен быть дополнен следующими функциями [Захаров и др., 1984]:

- структурированием и распараллеливанием программ;
- управлением параллельными вычислениями, в том числе параллельным выводом;
- управлением интерфейсом реляционных баз данных;
- проверкой типов и правильности данных.

### **6.3. Рефал-процессор**

*С. Л. Головкин, В. К. Смирнов*

Интерес к задачам символьной обработки вызвал стремление искать более дешевую реализацию языков, ориентированных на эту область.

Одним из языков символьной обработки, получивших широкое распространение в СССР, является язык Рефал (см. § 1.5). Он с успехом может быть использован для решения различных задач символьной обработки [Турчин, 1974].

Во второй половине 70-х годов в СССР были начаты работы по аппаратной реализации языка Рефал [Задыхайло и др., 1975]. Эта работа привела



к созданию в 1981 г. макета специализированного символьного процессора с входным языком Рефал [Мямлин и др., 1979]. В основу архитектуры процессора был положен промежуточный язык Рефал-компилятора, названный «языком сборки» [Романенко, 1987]. Аппаратной базой Рефал-процессора служил микропрограммный процессор, созданный в ИПМ АН СССР [Проскурии и др., 1976]. Опыт этой разработки был использован при создании специализированного процессора обработки текстовой информации ЕС-2702. Работа над процессором ЕС-2702 была завершена в 1985 г. [Myatlin et al., 1986].

### Назначение процессора

Процессор ЕС-2702 предназначен для эффективного выполнения программ, написанных на языке Рефал-2. Процессор совместим по входному языку с программными Рефал-системами на ЭВМ БЭСМ-6 и ЕС ЭВМ. В состав базового программного обеспечения входят компилятор, редактор связей, библиотека стандартных функций (включающая 69 функций) и программа обслуживания процессора (Монитор). Производительность процессора 5—8 тыс. шагов/с (2,2—2,5 млн. эквивалентных ЕС ЭВМ операций в секунду). Емкость оперативной памяти 1—3 Мбайт.

Процессор ЕС-2702 реализован в двух вариантах — как подключенный процессор и как виртуальный. Оба варианта имеют идентичные системы команд, но разные операционные системы (ОС), обеспечивающие разные режимы функционирования процессора. Если соединить через адаптер канал — канал машины ЕС-1035 с любой другой машиной серии ЕС, то центральный процессор машины ЕС-1035 может быть легко превращен в подключенный процессор ЕС-2702. Для этого не требуется никаких изменений в аппаратуре ЕС-1035, а только загрузка в его управляющую память микропрограммного обеспечения Рефал-процессора с магнитной ленты пультажного накопителя или с дисков путем запуска специальной процедуры на машине ЕС-1035. Дальнейшая работа с подключенным процессором производится через ЭВМ ЕС.

Если имеется только ЭВМ ЕС-1035, то на ней можно создать виртуальный процессор ЕС-2702. Для этого с помощью средств ОС ЕС в управляющую память (свободную от микропрограмм ЕС-1035) загружается специальная микропрограмма переключателя состояния процессора. После этого в любой момент времени ЭВМ ЕС-1035 может превратиться в процессор ЕС-2702 (за счет перезагрузки управляющей памяти, которую осуществляет переключатель состояния процессора), выполнить Рефал-программу, а затем вновь «стать» ЭВМ ЕС-1035.

Работа с подключенным и виртуальным процессором ЕС-2702 одинакова (рис. 6.18). Пользователь запускает программу обслуживания процессора (Монитор), указав в запуске набор данных, содержащий загрузочный модуль Ре-

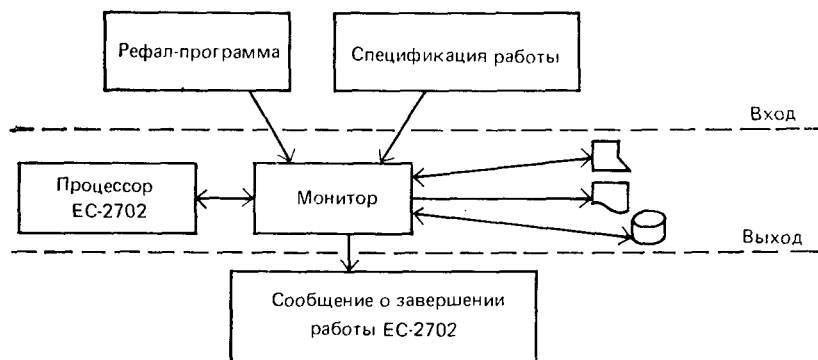


Рис. 6.18. Схема подключения процессора ЕС-2702

фал-программы, и набор данных, содержащий спецификации работы Рефал-машины. С помощью спецификаций пользователь может:

- управлять распределением памяти процессора;
- определять обработку особых состояний, возникающих при выполнении Рефал-программы;
- задать трассировку выполнения Рефал-программы;
- задать сбор статистических данных о выполненных Рефал-функциях или командах процессора.

Все дальнейшее обслуживание процессора (включая полную загрузку управляющей памяти, обслуживание ввода-вывода и т. п.) осуществляется программой Монитор.

По завершении выполнения Рефал-программы (нормального завершения или авоста) Монитор выводит соответствующее сообщение (в случае авоста сопровождаемое требуемой диагностической информацией) и также завершает свою работу.

### Архитектура процессора

С точки зрения архитектуры процессор ЕС-2702 представляет собой микропрограммно реализованную абстрактную Рефал-машину, которую обслуживает микропрограммная ОС. Реализацию операций абстрактной Рефал-машины (рис. 6.19) будем называть Рефал-процессором.

Функции микропрограммной ОС заключаются в инициализации процессора и завершении его работы, моделировании памяти Рефал-процессора [Головков, 1986], обслуживании запросов Рефал-процессора на выполнение операций ввода-вывода [Тульский, 1987], восстановлении при аппаратных сбоях, обработке особых состояний и т. д. Если микропрограммная ОС сама не может выполнить некоторое действие (например, обратиться к внешнему устройству, входящему в состав ЭВМ ЕС), то она передает сообщение программе Монитор, которая выполняет запрос.

Рефал-процессор представляет собой совокупность интерпретатора команд и набора запоминающих устройств (рис. 6.20).

Обрабатываемая информация размещается в поле зрения, которое представляет собой двусвязный список. Каждый элемент списка (звено) 12-байтовый. Из них 4 байта отводятся под информацию, а остальные 8 содержат две ссылки — на предыдущее и на следующее звено. Информационная часть звена имеет тег, который определяет тип информации, содержащейся в данном звене. Двусвязный список облегчает перемещение информации в поле зрения: перемещение некоторого выражения сводится к изменению ссылочных полей в крайних звеньях цепочки, которой представлено данное выражение. Это значительно ускоряет весь процесс переработки информации. Поле памяти содержит Рефал-программу, представляющую собой последовательность описаний функций. Каждое описание состоит из цепочки команд, анализирующих аргумент функции, и цепочки команд, строящих результат выполнения функции. Связь между полем зрения и полем памяти устанавливается с помощью стека рекурсий, каждый элемент которого содержит ссылку на некоторую функцию и на аргумент функции.

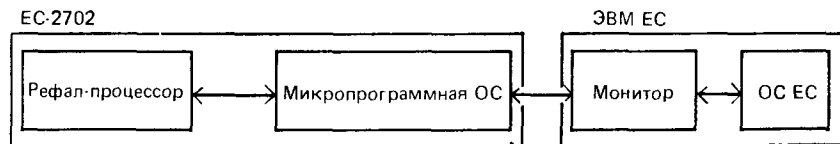


Рис. 6.19. Общая конфигурация абстрактной Рефал-машины

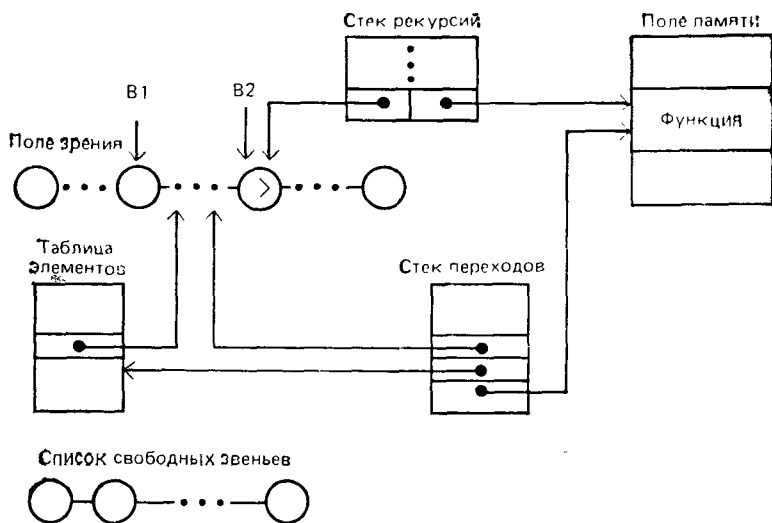


Рис. 6.20. Общая схема Рефал-процессора

Работа Рефал-процессора представляет собой последовательность обращений к функциям. Каждое обращение к функции — это шаг работы процессора. В каждом шаге две фазы — отождествление и замена. В начале каждого шага в регистры V1 и V2 заносятся начальный и конечный адреса участка поля зрения, содержащего аргумент функции. Регистры V1 и V2 являются внутренними регистрами процессора и программно недоступны. Фаза отождествления заключается в сопоставлении аргумента с образцом, представленным левыми частями предложений функции, и осуществляется последовательностью команд, порожденных компилятором из левых частей предложений этой функции.

Соответствие между аргументом и образцом отражается в таблице элементов. Сопоставив некоторому элементу образца определенную цепочку символов аргумента, команда заносит адреса этой цепочки в таблицу элементов. Регистры V1 и V2 содержат в каждый момент времени адреса левой и правой границ текущей «бреши» — участка аргумента, который требуется проанализировать. В результате выполнения команды происходит смещение одной из границ — либо V1 вправо, либо V2 влево. Когда происходит отождествление левой скобки, то правая граница V2 автоматически переставляется на парную правую скобку. Когда отождествляется правая скобка, то левая граница V1 автоматически переставляется на парную левую скобку. В результате таких перестановок в поле зрения могут возникнуть несколько «брешей». Границы всех «брешей» хранятся в таблице элементов. Анализ очередной «бреши» заканчивается, когда регистры V1 и V2 содержат адреса соседних элементов. После этого регистры V1 и V2 устанавливаются на границы следующей «бреши» и так до тех пор, пока существует хотя бы одна «брешь».

Если в процессе сопоставления с образцом возникает необходимость выбора альтернатив, то текущее состояние процесса сопоставления (адрес текущей команды, адрес текущего элемента таблицы элементов, текущее содержимое регистров V1 и V2) сохраняется в стеке переходов. Затем выбирается одна из альтернатив. Если в дальнейшем окажется, что выбранная альтернатива привела процесс сопоставления в тупик, то с помощью стека переходов восстанавливается состояние, предшествующее выбору альтернативы, и пробуются другая альтернатива. Необходимость выбора альтернатив возникает из-за наличия не-

Рис. 6.21. Схема интерпретатора команд

скольких предложений функции и различных возможных значений Е-переменных.

В случае успешного завершения отождествления процессор переходит к выполнению второй фазы шага — фазы замены, которая приводит к формированию результата, заменяющего аргумент в поле зрения.

Формирование результата осуществляется командами, порожденными компилятором из правых частей предложений функции.

Результат строится на списке свободных звеньев. Затем построенный результат отсоединяется от списка свободных звеньев и помещается в поле зрения вместо исходного аргумента, а исходный аргумент присоединяется к списку свободных звеньев. Если правая часть предложения содержит обращения к функциям, то это приводит к добавлению в стек рекурсий новых активационных записей.

Интерпретатор команд Рефал-процессора состоит из трех блоков NEXT, FAIL и STEP и набора микропрограмм-исполнителей команд (см. рис. 6.21). Блок NEXT выбирает очередную команду и ее аргументы, а затем запускает исполнитель этой команды. Команда может завершиться нормально или ненормально. В первом случае она возвращает управление блоку NEXT для выборки следующей команды, во втором — передает управление блоку FAIL. Блок FAIL пытается с помощью содержимого стека переходов вернуть Рефал-процессор в состояние, предшествующее последнему выбору альтернативы. Если это удастся, то управление возвращается блоку NEXT, в противном случае (стек переходов пуст) возникает авост.

Некоторые команды (завершающие шаг Рефал-процессора) передают управление блоку STEP, который анализирует стек рекурсий и определяет, есть ли еще функции, которые требуется выполнить. Если такие функции есть, то он помещает в регистр адреса команд адрес функции, в регистры В1 и В2 — адреса левой и правой границы аргумента функции и передает управление блоку NEXT. В противном случае блок STEP сигнализирует о том, что выполнение программы завершено.

Специализация архитектуры процессора позволила получить выигрыш в быстройдействии по сравнению с программной реализацией языка Рефал примерно на порядок. В табл. 6.3 приведены результаты прогона трех простых тестов на процессоре ЕС-2702 и на программной Рефал-системе на ЭВМ ЕС-1035:

ACKMN(3, 4) — программа вычисления функции Аккермана с аргументами 3, 4;

FIB(21) — программа вычисления 21-го члена последовательности Фибоначчи;

GREY(6) — программа порождения 6-разрядного кода Грея.

Процессор ЕС-2702 и ЭВМ ЕС-1035 имеют одну и ту же аппаратную базу, так что приведенные в таблице данные характеризуют тот выигрыш, который получается за счет отражения в архитектуре процессора специфики языка. Экви-

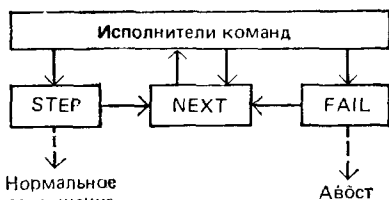


Таблица 6.3

Тест	Число шагов	Эквивалент шага	Быстрдействие, шаг/с	
			ЕС-1035	ЕС-2702
ACKMN(3,4)	25 408	316	549	7604
FIB(21)	88 552	234	762	7588
GREY(6)	221	280	539	5230

валент шага показывает число команд универсальной ЭВМ, затрачиваемых на выполнение одного шага.

**Подключенный процессор.** Подключенный процессор ЕС-2702 представляет собой самостоятельное устройство, имеющее блок сопряжения с универсальной ЭВМ. В этот блок входит селекторный канал и адаптер канала — канал. Через блок сопряжения процессор подключается к селекторному каналу универсальной ЭВМ.

Подготовка процессора ЕС-2702 к работе начинается с загрузки с пультавого накопителя в управляющую память ядра микропрограммной ОС. Процедура загрузки занимает около 30 с. После этого процессор переходит в состояние ожидания. Запуск процессора осуществляется программой Монитор через универсальную ЭВМ. В задании на запуск Монитора определяется Рефал-программа, которая должна выполняться на процессоре, и те наборы данных, которые используются при ее выполнении.

Работа начинается с того, что Монитор загружает в управляющую память процессора микропрограммы системы команд и недостающие модули ОС. Затем в оперативную память процессора загружается Рефал-программа, и процессор начинает ее выполнение. При выполнении программы процессор обращается к Монитору при необходимости выполнить операции ввода-вывода. По завершении Рефал-программы Монитор выводит результаты на внешние устройства универсальной ЭВМ и завершает свою работу, а процессор переходит в состояние ожидания нового задания.

Микропрограммное обеспечение подключенного процессора занимает около 44 Кбайт управляющей памяти. Кроме микропрограмм системы команд и ОС в этот объем входит ряд таблиц, располагающихся в управляющей памяти.

**Виртуальный процессор.** В режиме виртуального процессора аппаратура ЭВМ ЕС-1035 используется и для процессора ЕС-2702, и для универсальной ЭВМ. Это возможно, так как и ЕС-2702, и ЕС-1035 реализованы на базе одного и того же микропрограммного процессора ЕС-2635, разница заключается лишь в микропрограммах, содержащихся в памяти управления.

Подготовка ЭВМ ЕС-1035 к работе в режиме виртуального процессора заключается в загрузке в свободную от микропрограмм ЕС-1035 область управляющей памяти микропрограммы переключателя состояния процессора.

Работа в режиме виртуального процессора так же, как и в режиме подключенного процессора, начинается с запуска программы Монитор. Монитор формирует в оперативной памяти две области сохранения — ЕС и СП, в которых хранятся микропрограммы системы команд двух виртуальных процессоров: ЕС-1035 и ЕС-2702. Кроме того, в оперативной памяти выделяется область, которая используется в качестве оперативной памяти ЕС-2702. В СП-область загружаются микропрограммы процессора ЕС-2702. Затем Монитор загружает Рефал-программу в оперативную память ЕС-2702 и обращается к переключате-

Т а б л и ц а 6.4

**Микропрограммное обеспечение процессора ЕС-2702**

Компонент	Подключенный процессор		Виртуальный процессор	
	Число микро-команд	Доля от общего объема, %	Число микро-команд	Доля от общего объема, %
Система команд	3524	31,53	3524	58,04
Операционная система	7652	68,47	2443	40,23
Переключатель состояния процессора (ПСП)	—	—	105	1,73
Итого	11 176	100	6072	100

лю состояния процессора. Переключатель состояния процессора запоминает в ЕС-области текущее содержимое управляющей памяти, а содержимое СП-области переписывает в управляющую память. В результате происходит переключение из состояния ЕС в состояние СП, т. е. процессор ЕС-2635 начинает функционировать как ЕС-2702 и выполняется Рефал-программа, загруженная в оперативную память ЕС-2702.

Когда в процессе выполнения Рефал-программы требуется доступ к внешним устройствам, процессор ЕС-2702 обращается к переключателю состояния процессора, происходит смена содержимого управляющей памяти и процессор ЕС-2635 переходит в ЕС-состояние. Управление получает программа Монитор, которая выполняет необходимые действия, например считывание данных с внешнего устройства в оперативную память процессора ЕС-2702, и возвращает управление переключателю состояния процессора. Происходит перезагрузка управляющей памяти, и процессор ЕС-2702 продолжает работу.

Микропрограммное обеспечение виртуального процессора занимает 24 Кбайт управляющей памяти.

Каждый из этих режимов имеет свои достоинства и недостатки (см. табл. 6.4). Виртуальный процессор ЕС-2702 может быть создан на любой ЭВМ ЕС-1035. Для подключенного процессора ЕС-2702 требуется отдельный процессор ЕС-2635 с оперативной памятью и адаптером канал — канал. Кроме того, ОС универсальной ЭВМ, к которой подключен процессор ЕС-2702, должна обладать средствами обслуживания адаптера канал — канал.

В режиме виртуального процессора ЭВМ ЕС-1035 практически полностью занята обслуживанием процессора ЕС-2702, так как ее процессор ЕС-2635 работает либо как ЕС-процессор, либо как СП-процессор. В распоряжении процессора ЕС-2702 при этом находится не вся оперативная память, входящая в состав ЕС-1035, так как часть ее занята ОС и программой Монитор.

## **Заключение**

Проблема разработки программных и аппаратных средств поддержки интеллектуальных систем — важнейшее научно-техническое направление исследований, ориентированное на повышение эффективности и качества создаваемых систем обработки знаний. В этом направлении пока что сделаны лишь первые шаги. На успешное решение проблемы существенное влияние оказывает состояние исследований в следующих важных областях: разработка моделей и методов обработки знаний, создание параллельных алгоритмов и методов организации этой обработки, развитие и совершенствование средств микропроцессорной техники и соответствующей технологии их производства.

В данном томе нашли отражение лишь некоторые вопросы из множества перспективных, которые стоило бы рассмотреть в рамках отмеченных научных направлений. Отдельные разделы изложены довольно концептивно и не дают возможности получить полное представление по рассматриваемой проблеме. Отсутствие многих материалов (например, специализированных процессоров для обработки изображений, речи и др.) не означает, что исследования в данной области не ведутся вовсе. Интеллектуальные системы в настоящее время настолько бурно развивающаяся область знаний, а влияние развивающейся микроэлектронной технологии настолько ощутимо, что отразить достигнутый уровень исследований по всем важнейшим направлениям в справочном пособии чрезвычайно сложно, а дать в какой бы то ни было степени объективный прогноз даже на ближайшие годы, по-видимому, безнадежное дело.

Кто может предсказать последствия результатов исследований в области

разработки новых поколений микропроцессоров, нейронных сетей и нейрокомпьютеров, систем на базе новых приборов субмикронной технологии? Прорыв в этих и других направлениях можно ожидать в ближайшие годы, и это незамедлительно скажется на реализации алгоритмов обработки знаний в интеллектуальных системах.

Отличительной особенностью современных исследований в области создания программных и аппаратных средств поддержки информационных процессов в интеллектуальных системах является внедрение полученных теоретических результатов в промышленное производство соответствующих изделий. Ряд западных фирм приступил к выпуску на рынок широкого ассортимента специальных блоков, используемых в качестве средств поддержки программного обеспечения в таких сферах промышленного производства, как станкостроение, автомобилестроение, самолетостроение, автоматизированное производство, ядерная энергетика, биомедицинское оборудование и др. Эти блоки существенно влияют на сокращение стоимости и сроков разработки управляющих систем, а также на повышение надежности и эффективности их работы. Подключение к персональным ЭВМ и рабочим станциям аппаратных средств поддержки программного обеспечения в значительной мере позволяет повысить эффективность их использования, а также возможности реализованных с их помощью логических, информационно-поисковых и других систем, включая системы автоматизированного проектирования и конструирования. На повестке дня стоит вопрос о разработке устройств, ориентированных на применение их в составе интеллектуальных, в частности экспертных, систем.

Оценивая в общих чертах состояние дел в области разработки специальных средств поддержки для интеллектуальных систем, можно отметить следующее. Эта область исследований пока еще находится в стадии становления. Накопленный опыт реализации алгоритмов обработки знаний еще недостаточен, чтобы можно было ставить вопрос о типизации и унификации соответствующих аппаратных устройств. Однако исследования в этом направлении ведутся, и в ближайшее время можно ожидать появления новых ощутимых результатов.

# СПИСОК ЛИТЕРАТУРЫ

- Абдрахманов, 1983 Абдрахманов А.А. Система программирования ОЛИСП // Системное и теоретическое программирование: Тез. докл. на IV Всесоюз. симп. - Кишинев: Штиинца, 1983. - С. 3-4.
- Алешин и др., 1986 Алешин А.Ю., Шерстнев В.Ю. Система программирования РЕФАЛ/2 для персональных ЭВМ // Автоматизация производства систем программирования: Тез. докл. III Всесоюз. конф. - Таплин, 1986. - С. 50-52.
- Альбрехт, 1978 Альбрехт А. О схемах из клеточных элементов // Проблемы кибернетики. - 1978. - N 33.
- Ангелова и др., 1984 Ангелова Г. и др. Языки программирования для искусственного интеллекта // Представление знаний в человеко-машинных и робототехнических системах. Т. В. Инструментальные средства разработки систем, ориентированных на знания/ ВЦ АН СССР и ВИНТИ. - М., 1984. - С. 31-72.
- Ангер, 1977 Ангер С. Асинхронные последовательностные схемы. - М.: Мир, 1977. - 400 с.
- Артемьева и др., 1983 Артемьева И.Л. и др. Инструментальный комплекс для реализации языков представления знаний // Программирование. - 1983. - N 3. - С. 78-89.
- Артемьева и др., 1984 Артемьева И.Л. и др. Пересмотренное сообщение о реляционном языке программирования // Языки представления знаний и вопросы реализации экспертных систем / ДВНЦ АН СССР. - Владивосток, 1984. - С. 99-122.
- Асратян и др., 1976 Асратян Р.Э., Волков А.Ф., Лысиков В.Т. Об одном подходе к аппаратной реализации // Управляющие системы и машины (УСМ). - 1976. - N 3. - С. 27-31.
- Асратян и др., 1978 Асратян Р.Э., Волков А.Ф., Лысиков В.Т. Микропроцессорная реализация команд для обработки текстов программ // УСМ. - 1978. - N 2. - С. 26-32.
- Байдун, 1980 Байдун В.В. О реализации языка FRL на ЕС ЭВМ. // Представление знаний в системах искусственного интеллекта/ МДНТП. - М., 1980.
- Базисный, 1977 Базисный Рефал и его реализация на вычислительных машинах/ ЦНИПИАСС. - М., 1977. - 238 с.
- Барздин, 1965 Барздин Я.М. Емкость среды и поведение автоматов // ДАН СССР. - 1965. - Т. 160, N 2.
- Барздин, 1966 Барздин Я.М. Моделирование логических сетей на автоматах Неймана-Черча // Проблемы кибернетики. - 1966. - N 17.
- Барни, 1985 Барни К. Арсенид-галлиевый микропроцессор с тактовой частотой 200 МГц // Электроника. - 1985. - Т. 58, N 11. - С. 11-12.
- Баронец, 1986 Баронец В.Д. Проектирование лингвистических устройств на основе нечеткой логики // Автоматика и тепломеханика. - 1986. - N 12.



- Баррон и др.,  
1983
- Барштейн и др.,  
1983
- БИНТИ ТАСС,  
1988а
- БИНТИ ТАСС,  
1988б
- Большакова и др.,  
1986
- Брябрин,  
1988
- Бычков и др.,  
1982
- Вагин,  
1986
- Вагин и др.,  
1987
- Вайтершиц и др.,  
1984
- Варшавский и др.,  
1973
- Варшавский,  
1976
- Варшавский,  
1986
- Варшавский и др.,  
1988
- Васильев и др.,  
1985
- Введенев,  
1975
- Баррон И., Кэвип П., Мэй Д., Вильсон П. Транспьютер с быстродействием 5 млн. операций/секунду и более // Электроника.-1983.-N 23.-С.26-35.
- Барштейн Л.С., Мелихов А.Н. Формирование классов расплывчатых ситуаций специализированным устройством оцувствленного робота // Изв. АН СССР. Техн. кибернетика. - 1983. - N 4.
- Нейронные ЭВМ// Бюллетень иностранной научно-технической информации (БИНТИ) ТАСС.- 20 апреля 1988.- N 16.- С. 17-18.
- Цифровые процессоры обработки сигналов //БИНТИ ТАСС. - 20 апреля 1988. - N 16.- С. 18-27.
- Большакова Е.И., Мальковский М.Г., Пильщиков В.Н. Обучающая система ЛУЧ // Научно-технические статьи по математике. - М.: Высшая школа, 1986.- Вып. 13. - С. 151-163.
- Брябрин В.М. Программное обеспечение персональных ЭВМ.- М.: Наука, 1988.- 271 с.
- Бычков С.П. и др. Компилятор с языка СИМУЛА-67 для ЕС ЭВМ.- Препринт.- М., 1982.- 42 с.- (ИПМ АН СССР).
- Вагин В.Н. Параллельная дедукция на семантических сетях// Изв. АН СССР. Техн. кибернетика.- 1986.- N 5.
- Вагин В.Н., Захаров В.Н., Розенблюм Л.Я. К логическому выводу на сетях Петри// Изв. АН СССР. Техн. кибернетика.-1987.- N 5.
- Вайтершиц М., Захаров В.Н., Эйсымонт Л.К. Программно-аппаратные методы параллельной обработки//Представление знаний в человеко-машинных и робототехнических системах. Т.В. Инструментальные средства разработки систем, ориентированных на знания/ ВЦ АН СССР, ВИНТИ.- М., 1984.- С.188-204.
- Варшавский В.И., Мараховский В.Б., Песчанский В.А., Розенблюм Л.Я. Однородные структуры: Анализ. Синтез. Поведение. - М.: Энергия, 1973.
- Апериодические автоматы/ Под ред. В.И. Варшавского.- М.: Наука, 1976.- 424 с.
- Автоматное управление асинхронными процессами в ЭВМ и дискретных системах / Под ред. В.И.Варшавского.- М.: Наука, 1986.- 400 с.
- Варшавский В.И. и др. Модели для спецификации и анализа асинхронных процессов в схемах // Изв. АН СССР. Техн.кибернетика.- 1988.- N 2.
- Васильев В.В., Кузьмин В.В. Построение алгоритмически однородных многопроцессорных систем//ДАН УССР, - 1985.- Т. А, N 6.
- Введенев А.А. Рефал-компилятор для ДОС ЕС // Теория систем программирования и исследование операций.- 1975.- N 8.- С.31-43.

- Вулф,  
1987
- Вулф А. Модули на базе однокристалльного Лисп-процессора для ЭВМ военного назначения // Электроника. - 1987. - Т. 60, N 5. - С. 27-3.
- Гаврилова,  
1987
- Гаврилова Т.А. Извлечение знаний: Психолингвистический аспект // Технология разработки экспертных систем: Тез. докл. республик. школы-семинара. - Кишинев, 1987, - С.44-46.
- Гинзбург,  
1968
- Гинзбург С.А. Математическая непрерывная логика и изображение функций.- М.:Энергия, 1968.
- Глазунов и др.,  
1968
- Глазунов Н.И., Горяшко А.П. Асимптотические оценки сложности логических сетей в физических средах // Изв. АН СССР. Техн. кибернетика, -1968. - N 5.
- Глухи и др.,  
1987
- Глухи Л., Захаров В.Н., Кочиш И. Распределенные системы обработки данных на базе однородных спецпроцессоров // Изв. АН СССР. Техн. кибернетика.- 1987.-N 4.
- Глушков и др.,  
1975
- Глушков В.М. и др. Технология программирования и проблемы ее автоматизации //УСиМ.- 1975.- N 6.- С.75-93.
- Глушков и др.,  
1977
- Глушков В.М. и др. АНАЛИТИК (алгоритмический язык для описания вычислительных процессов с использованием аналитических преобразований) //Кибернетика.-1977.-N 3 .-С.102-134.
- Головкин,  
1980
- Головкин Б.А. Параллельные вычислительные системы.- М.: Наука, 1980.- 520 с.
- Головков и др.,  
1982
- Головков С.Л., Наумов Н.А., Смирнов В.К. О некоторых новых средствах языка рекурсивных функций.- Препринт.- М., 1982.-(ИПМ АН СССР, N 6).
- Головков,  
1986
- Головков С.Л. Управление памятью специализированного процессора ЕС 2702.- Препринт.- М., 1986. - (ИПМ АН СССР, N 11).
- Горбачев,  
1984
- Горбачев С.Б. Метод формального описания языка представления знания МЕДИФОР-3 //Языки представления знаний и вопросы реализации экспертных систем/ДВНЦ АН СССР. - Владивосток, 1984.- С. 4В-5В.
- Гордиенко и др.,  
1985
- Гордиенко Е.К., Захаров В.Н. Управление процессами в базах знаний//Изв. АН СССР. Техн. кибернетика.- 1985.- N 5.-С.175-193.
- Гордиенко,  
1986
- Гордиенко Е.К. Реализация поисковых функций языка FRL с применением двухтеговой ассоциативной памяти // Изв. АН СССР. Техн.кибернетика. - 1986. - N 2.- С. 71-88.
- Гордиенко и др.,  
1986
- Гордиенко Е.К., Захаров В.Н., Миронов А.Ю. Реализация параллельных алгоритмов логического вывода в матричной однородной среде // Изв.АН СССР. Техн. кибернетика.- 1986.- N 5.
- Горяшко,  
1972
- Горяшко А.П. Диффузионная модель функционирования вероятностного автомата // Изв. АН СССР. Техн. кибернетика.- 1972. - N 4.
- Горяшко и др.,  
1988
- Горяшко А.П., Шура-Бура А.Э.Методы оценки отказоустойчивых структур СБИС // Изв.АН СССР. Сер. Техн.кибернетика.-1988.-N 6.-С.133-142.
- Грисуолд и др.,  
1980
- Грисуолд Р., Поудис Дж., Полонски И. Язык программирования СНОБОЛ-4 - М.:Мир, 1980, 268 с.

- Громов,  
1985  
Грох и др.,  
1983  
Дал и др.,  
1969  
Дейт,  
1980  
Джой,  
1987  
Дириг,  
1987  
Евреинов и др.,  
1966  
Евреинов,  
1981  
Ежкова и др.,  
1977  
Ефимов Е.И.  
1982  
Загадская и др.,  
1980  
Загоруйко и др.,  
1986  
Загоруйко,  
1987  
Заде,  
1976  
Задыхайло и др.,  
1974  
Задыхайло и др.,  
1975  
Громов Г.Р. Национальные информационные ресурсы.- М.: Наука, 1985.- 241 С.  
Грох А.В. и др. Анализ метаязыковых описаний символьных преобразований на основе смешанных вычислений // Прогрессивные технологии программирования/ МДНТП.- М., 1983.- С. 68-75.  
Дал У. И., Мюрхауг Б., Нюгорд К. СИМУЛА-67. Универсальный язык программирования.- М.: Мир, 1969. - 169с.  
Дейт К. Введение в системы баз данных.- М.: Наука.- 1980.-463 с.  
Джой Б. Сомнений нет - архитектура RISC победила//Электроника.- 1987.- Т.60, N 18.- С. 32-33.  
Дириг М.Ф. Архитектура машин для искусственного интеллекта// Реальность и прогнозы искусственного интеллекта.-М.:Мир, 1987.- С. 209-230.  
Евреинов Э.В., Косарев Ю.Г. Однородные вычислительные системы высокой производительности.-М.: Радио и связь, 1966.  
Евреинов Э.В. Однородные вычислительные системы, структуры и среды. - М.: Радио и связь, 1981.  
Ежкова И.В., Поспелов Д.А. Принятие решений при нечетких основаниях. / I. Универсальная шкала // Изв. АН СССР. Техн. кибернетика.- 1977.- N 6.  
Ефимов Е.И. Решатели интеллектуальных задач.- М.: Наука, 1982.- 316 с.  
Загадская Л.С., Лозовский В.С., Сокольников А.И., Горячук В.Ф., Реализация базовых процессов в системе ситуационного управления // Вопросы кибернетики. Ситуационное управление. Теория и практика /Под ред. Д.А. Поспелова.- М., 1980.- С. 94-108.  
Загоруйко Ю.А., Неустроев А.Л., Неверов И.В., Гринберг С.Я. Технопогический пакет для конструирования средств работы с семантическими сетями // Системы обработки знаний: Тр. Междунар. рабочей конф. по комплексным научным проектам КНП-1 и КНП-2.Т. ИИ. - Смоленце, ноябрь 1986. - С. 23-28.  
Загоруйко Ю.А. Технопогия конструирования средств обработки знаний на основе семантических сетей. Общая схема и базовые средства. - Препринт.- Новосибирск, 1987. - 30с. - (ВЦ СО АН СССР, N 749.)  
Заде Л. Понятие лингвистической переменной и его применение к принятию приближенных решений.-М.:Мир, 1976.  
Задыхайло И.Б., Мямлин А.Н., Смирнов В.К., Эйсмонт Л.К. Об эффективной аппаратной реализации языка для описания объектов на уровне понятий и символьных преобразований// Искусственный интелект. Итоги и перспективы/ МДНТП.- М., 1974.- С. 157-165.  
Задыхайло И.Б. и др. О повышении эффективности символьных преобразований. - Препринт. - М., 1975. - (ИПМ АН СССР, N 15).

- Задыхайло и др.,  
1979а Задыхайло И.Б., Садыхов Я.А., Мельников Б.М. Проект ассоциативного процессора на ЦМД, ориентированного на поддержку реляционных баз данных.- Препринт.- М., 1979.- 24 с.-(ИПМ АН СССР, N 180 ).
- Задыхайло и др.,  
1979б Задыхайло И.Б., Садыхов Я.А. Алгоритм реализации запроса в виде  $\lambda$ -выражения, допускающий массовые параллельные операции.- Препринт.- М., 1979. - (ИПМ АН СССР, N 18).
- Задыхайло и др.,  
1980 Задыхайло И.Б. и др. Исследование процессов параллельного выполнения компилирующих программ некоторого типа.- Препринт.-М., 1980.- (ИПМ АН СССР, N 124).
- Захаров и др.,  
1984 Захаров В.Н. и др. Специализированные вычислители и аппаратные средства// Представление знаний в человеко-машинных и робототехнических системах, Т.В. Инструментальные средства разработки систем, ориентированных на знания / ВЦ АН СССР, ВИНТИ.- М., 1984.- С. 166-187.
- Йодан,  
1979 Йодан Э. Структурное проектирование и конструирование программ: Пер. с англ. - М.: Мир, 1979, -415с.
- Каляев,  
1984 Каляев А.В. Многопроцессорные системы с программируемой архитектурой.- М.: Радио и связь, 1984.
- Кандрашина,  
1986 Кандрашина Е.Ю. Средства представления информации о времени в базах знаний. Последовательности событий // Изв. АН СССР, Техн. кибернетика, - 1986.- N 5. - С. 211-231.
- Кафаров и др.,  
1968 Кафаров В.В. и др. Принципы описания химико-технологических процессов с помощью нечетких множеств//ДАН СССР, 1978.- Т.243, N 1.
- Кахро и др.,  
1981 Кахро М.И., Капья А.П., Тыгу Э.Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ).-М.: Финансы и статистика, 1981. - 158 с.
- Клещев и др.,  
1978 Клещев А.С. и др. МЕДИФОР - язык представления медицинского диагностического знания. Методология программирования. - Препринт.- Владивосток, 1978.- 41 с.-(ИАПУ ДВНЦ АН СССР).
- Клименко и др.,  
1981 Клименко В.П., Погребинский С.Б., Фишман Ю.С. Особенности программно-ориентированных комплексов на базе ЭВМ СМ-1410 // АСУ: Проблемно-ориентированные комплексы/ Ин-т кибернетики АН СССР.- Киев, 1981. - С. 26-32.
- Климов и др.,  
1972 Климов Анд.В., Романенко С.А., Турчин В.Ф. Компилятор с языка РЕФАЛ.- Препринт. - М., 1972.- 37 с.- (ИПМ АН СССР).
- Климов и др.,  
1973 Климов Анд.В., Романенко С.А., Турчин В.Ф. Теоретические основы синтаксического отождествления.- Препринт.- М., 1973.- 75 с. - (ИПМ АН СССР).
- Климов и др.,  
1974 Климов Анд.В., Романенко С.А., Травкина Е.В. Инструкция по работе с мониторной системой Рефал для БЭСМ-6.- Препринт.- М., 1974. - 43 с. - (ИПМ АН СССР).

- Климов и др., 1987  
Климов Анд.В., Романенко С.А. Система программирования РЕФАЛ-2 для ЕС ЭВМ. Описание входного языка.- Препринт.- М., 1987.- 53 с.- (ИПМ АН СССР).
- Клоксин и др., 1987  
Клоксин У., Меллиш К. Программирование на языке ПРОЛОГ. - М.:Мир, 1987.- 479 с.
- Кодачигов, 1984  
Кодачигов В.И. Об управлении коммуникацией процессоров, реализуемых в МВС с перенастраиваемой архитектурой// Многопроцессорные вычислительные структуры.- 1984.- Т.6.
- Корнеев, 1985  
Корнеев В.В. Архитектура вычислительной системы с программируемой структурой// Многопроцессорные вычислительные структуры.-1985.-Т. 8.
- Коршунов, 1967  
Коршунов А.Д. Об оценках сложности схем из объемных функциональных элементов// Проблемы кибернетики.- 1967.- N 19.
- Котов, 1984  
Котов В.Е. Сети Петри.- М.: Наука, 1984. -160 с.
- Кофман, 1982  
Кофман А. Введение в теорию нечетких множеств.-М.: Радио и связь, 1982.
- Кох и др., 1981  
Кох Д., Хайкинг В. Анализ немецких предложений с помощью модифицированной АТН-системы // Лингвистические процессоры и представление знаний /Под ред. А. С. Нариньяни. - Новосибирск: ВЦ СО АН СССР.- 1981. - С. 95-128.
- Кохонен, 1982  
Кохонен Т. Ассоциативные запоминающие устройства.- М.:Мир, 1982.- 384 с.
- Куприянов и др., 1983  
Куприянов М.С., Виноградов В.Б., Бялый В.О. Микропроцессорная реализация лингвистического терминала// Микропроцессорные системы / ЛДНТП.- Л.,1983.- С.23-30.
- Лазарев и др., 1984  
Лазарев В.Г., Пилю Е.И., Турута Е.Н. Построение программируемых управляющих устройств.- М.: Энергоатомиздат, 1984.
- Левин и др., 1986  
Левин Д.Я. и др. Система программирования на основе виртуального процессора ГАММА-1. - Препринт.- Новосибирск, 1986. -30 с. - (ВЦ СО АН СССР, N 669).
- Липаев и др., 1983  
Липаев В.В., Серебровский Л.А., Гаганов П.Г. и др. Технология проектирования комплексов программ АСУ.- М.: Радио и связь, 1983.- 264 с.
- Лихолип, 1985а  
Лихолип В. Н. Язык ПЛЭНЕР-ЭЛЬБРУС // Автоматизация и роботизация производства с применением микропроцессорных средств/ИМ АН МССР. - Кишинев, 1985.- С. 63-71.
- Лихолип, 1985б  
Лихолип В. Н. Реализация системы ПЛЭНЕР-ЭЛЬБРУС // Логико-комбинаторные методы в искусственном интеллекте и распознавании образов/ ИМ АН МССР. - Кишинев, 1985.- С. 37-45.
- Лозовский В., 1978  
Лозовский В.С. Задание реляционной базы данных в виде мультисети и реализация поиска по образцу //Информационное и программное обеспечение систем ситуационного управления.- Препринт.- Киев, 1978.- С. 13-24.- ( ИК АН УССР, N 78-14 ).

- Лозовский В.,  
1979 Лозовский В.С. Ситуационная и дефиниторная семантика системы представления знаний // Кибернетика.- 1979.- N 2.- С. 98-101.
- Лозовский В.,  
1981 Лозовский В.С. СУБД REX для реализации интегрированных семиотических моделей // Представление знаний: Труды IX Всесоюз. симп. по кибернетике (Сухуми, 10-15 ноября 1981 г.).- Т.1 / Научный Совет по комплексной проблеме "Кибернетика", ИК АН ГССР, ВЦ АН СССР. - М., 1981.- С. 54-56.
- Лозовский В.,  
1984 Лозовский В.С., Иммедиа-эффектное программирование и задача интерпретации ситуаций // Тез. докл. конф. КНВВТ АН социалистических стран "Проблемы искусственного интеллекта и распознавания образов". Секция 1. - Киев, 1984.- С. 113-115.
- Лозовский С.,  
1985 Лозовский С.В. Язык видеоформ FL и его использование в диалоговых системах // Диагностический Человек - ЭВМ: Труды IV Всесоюз. конф. Ч.1.- Киев, 1985.- С. 63-65.
- Майерс,  
1985 Майерс Г. Архитектура современных ЭВМ.- Кн. 2.- М.: Мир, 1985.- 308 с.
- Мапьяковский,  
1985 Мапьяковский М.Г. Диалог с системой искусственного интеллекта. - М.: МГУ, 1985. - 214 с.
- Мансуров и др.,  
1987а Мансуров Н.Н., Эйсымонт Л.К. Реализация расширенного языка Рефал на односвязных списках с кольцевыми цепочками. - Препринт. - М., 1987. - (ИПМ АН СССР, N 20).
- Мансуров и др.,  
1987б Мансуров Н.Н., Эйсымонт Л.К. Реализация языка Рефал на односвязных списках с кольцевыми цепочками. - Препринт. - М., 1987. - (ИПМ АН СССР, N 203).
- Мануэль,  
1985 Мануэль Т. Попытки внедрения экспертных систем и проблемы интеграции // Электроника.- 1985.- Т.58, N 14.- С. 27-38.
- Мануэль,  
1987а Мануэль Т. Активные поиски путей повышения быстродействия компьютеров // Электроника. - 1987. - Т. 60, N 18.- С.18-22.
- Мануэль,  
1987б Мануэль Т. Десятикратное увеличение быстродействия компьютеров с архитектурой Нурегcube//Электроника. - 1987.-Т. 60, N 18.- С. 63-64.
- Мануэль,  
1987в Мануэль Т. Лисп-машины фирмы TI обеспечивают пятикратный выигрыш в скорости решения задач искусственного интеллекта // Электроника. - 1987.- Т.60, N 13.- С. 65-66.
- Мануэль,  
1987г Мануэль Т. Процессор, сочетающий возможности обработки числовых и символьных данных // Электроника. - 1987.- Т.60, N 8.- С. 7-9.
- Мануэль,  
1987д Мануэль Т. Супермикрокомпьютер с архитектурой CISC и вычислительной мощностью крупной ЭВМ // Электроника.- 1987.- Т. 60, N 18. - С.35-39.
- Марчук и др.,  
1980 Марчук Г.И., Котов В.Е. Модульная асинхронная развиваемая система// Параллельное программирование и высокопроизводительные системы/ ВЦ СО АН СССР.- Новосибирск, 1980.- с. 145-158.

- Маршалл,  
1986 Маршалл М. Начало промышленного выпуска компьютеров, работающих на языке Лисп// Электроника. - 1980.- N 20.- С. 75-78.
- Многопроцессорные,  
1975 Многопроцессорные вычислительные системы/Под ред. В.Л. Арлазарова, А.Ф. Волкова. -М.: Наука, 1975.-143 С.
- Мелихов и др.,  
1979 Мелихов А.Н., Берштейн Л.С. Принципы построения однородных структур для реализации расплывчатых алгоритмов / Прикладные аспекты теории автоматов.Т.1. - Варна, 1979.
- Мелихов и др.,  
1981 Мелихов А.Н., Берштейн Л.С. Конечные четкие и нечеткие множества. Ч. 2. Нечеткие множества.- Таганрог: ТРТИ, 1981.
- Мелихов и др.,  
1985 Мелихов А.Н., Берштейн Л.С., Коровин С.Я. Сжатие множества эталонных ситуаций в лингвистических моделях ситуационного управления//Автоматика и телемеханика.-1985.-N 2.
- Мелихов и др.,  
1986а Мелихов А.Н., Берштейн Л.С., Коровин С.Я. Расплывчатые ситуационные модели принятия решений.- Таганрог:ТРТИ, 1986.
- Мелихов и др.,  
1986б Мелихов А.Н. и др. Лингвистический вычислительный комплекс для обработки расплывчатой информации // Изв. СКНЦВШ. Техн. науки, 1986.- N 2.
- Миллер,  
1971 Миллер Р.Е. Теория переключательных схем. Т. 2-М.: Наука, 1971.- 304 с.
- Миллс,  
1970 Миллс Х. Программирование больших систем по принципу сверху вниз //Средства отладки больших систем. -М.: Статистика, 1970. - С. 41-56.
- Минц,  
1986 Минц Г.А. Полное исчисление для чистого Пролога. - Изв. АН СССР, 1986, N 3, стр. 27-42.
- Мото-ока,  
1984 ЭВМ пятого поколения. Концепции, проблемы, перспективы / Под ред. Мото-ока Т.: Пер. с англ. - М.: Финансы и статистика, 1984. - 109 с.
- Мур,  
1966 Мур Э.Ф. Математические модели самовоспроизведения/ Математические проблемы в биологии.- М.: Мир, 1966.
- Мямлин и др.,  
1979 Мямлин А.Н. и др. Специализированный символьный процессор // Технология программирования: Тез. докл. / Ин-т кибернетики АН УССР.- Киев, 1979.
- Мямлин и др.,  
1988 Мямлин А.Н.,Рубин А.Г.,Смирнов В.К., ЛИСП-процессоры- аппаратная база систем искусственного интеллекта - Препринт.- М., 1988.- 37 с.- (ИПМ АН СССР, N 57).
- Нариньяни,  
1986 Нариньяни А.С. Недоопределенность в системах представления и обработки знаний //Изв. АН СССР. Техн. кибернетика. - 1986. - N.5. - С. 3-28.
- Нариньяни и др.,  
1986 Нариньяни А.С., Кандрашина Е.Ю. Технологический комплекс построения баз знаний // Вычислительные системы и программное обеспечение. - Новосибирск, 1986. - С. 100-118.
- Нариньяни и др.,  
1987 Нариньяни А.С., Загорюлько Ю.А. Простые средства управления активацией для производственных систем. // Информатика. Технологические аспекты. - Новосибирск, 1987. - С. 118-126.
- Нечеткие  
1986 Нечеткие множества в моделях управления и искусственного интеллекта / Под ред. Д.А. Поспелова.- М.:Наука, 1986.

- Нильсон, 1985  
Офман, 1965  
Пильщиков, 1982  
Пильщиков, 1983  
Питерсон, 1984  
Подколзин, 1975  
Поздняков, 1977  
Прадхан, 1986  
Проскурин и др., 1976  
Розенблум, 1983  
Романенко, 1987а  
Романенко, 1987б  
Сапаты, 1986  
Сергиевский, 1986  
Сердобольский, 1969  
Смит, 1987  
Тербер, 1985  
ТИИЭР, 1986  
Тулский, 1987  
Турчин, 1966  
Турчин, 1968
- Нильсон Н. Принципы искусственного интеллекта.- М.: Мир, 1985.  
Офман Ю.П. Универсальные автоматы. - М., 1965. - (Тр. Моск. матем. о-ва. 1965.- Т.14).  
Пильщиков В. Н. Система программирования ПЛЭНЕР-БЭСМ. -М.: МГУ. - 1982. - 56 с.  
Пильщиков В.Н. Язык ПЛЭНЕР. - М., Наука, 1983, 208 с.  
Питерсон Дж. Теория сетей Петри и моделирование систем.-М.:Мир, 1984.- 264 с.  
Подколзин А.П. Сложность моделирования в однородных средах // Проблемы кибернетики.- 1975.- N 30.  
Поздняков Л.А. Сравнение двух схем использования параллельного доступа к памяти при работе с перемешанными таблицами. - Препринт.- М., 1977.- (ИПМ АН СССР, N 96).  
Прадхан Д.Н. Устойчивые к отказам архитектуры мультипроцессоров // ТИИЭР.- 1986.- Т.74.- N 5.  
Проскурин М.И., Смирнов В.К., Юдина М.Л. Микропрограммный процессор.- Препринт.- М., 1976.- (ИПМ АН СССР, N 27).  
Розенблум Л.Я. Сети Петри // Изв. АН СССР Техн. кибернетика.- 1983.- N 5.- С.12-40.  
Романенко С.А. Реализация Рефал-2. - Препринт./ ИПМ АН СССР. - М., 1987.  
Романенко С.А. Рефал-4 - расширение Рефала-2, обеспечивающее выразимость результатов прогонки.- Препринт./ ИПМ АН СССР. - М., 1987.- 33 с.  
Сапаты П.С. Язык ВОЛНА-0 как основа навигационных структур для баз знаний на основе семантических сетей // Изв. АН СССР. Техн.кибернетика; 1986.- N 5.  
Сергиевский Г.М. Построение экспертных систем на основе концепции расширяемого языка. // Экспертные системы/МДНТП.- М.,1986,-С. 130-134.  
Сердобольский В.И. Язык РЕФАЛ и его использование для преобразования алгебраических выражений. //Кибернетика.- 1969.- N 3.- С. 45-51.  
Смит Ф. Архитектура CISC еще способна удержать свои позиции // Электроника.- 1987.- Т.60, N 18.- С. 34-35.  
Тербер К.Дж. Архитектура высокопроизводительных вычислительных систем.- М.: Наука, 1985.- 272 с.  
ТИИЭР. - 1986. - Т. 74, N 7. (Обработка естественных языков. ) - 180 с.  
Тулский В.П. Средства связи символического процессора ЕС 2702 с универсальной ЭВМ.- Препринт.-М.,1987.- (ИПМ АН СССР, N 169).  
Турчин В.Ф. Метаязык для формального описания алгоритмических языков// Цифровая вычислительная техника и программирование.- М.: Сов. радио, 1966.- С. 116-124.  
Турчин В.Ф. Метаалгоритмический язык // Кибернетика. - 1968, - N 4 - С. 45-54.



- Турчин, 1971 Турчин В.Ф. Программирование на языке Рефал. Ч.2. Формальное описание и принципы реализации Рефапа. - Препринт. - М., 1971. - 60 с. (ИПМ АН СССР, N 43).
- Турчин, 1972 Турчин В.Ф. Описание аналитических преобразований с помощью рекуррентных соотношений в рамках языка РЕФАЛ // Вычислительная математика и вычислительная техника. - Харьков. - 1972. - N 3. - С. 28-29.
- Турчин, 1974а Турчин В.Ф. Базисный РЕФАЛ. Описание языка и основные приемы программирования. - М.: ЦНИПИАС, 1974. - 258 с.
- Турчин, 1974б Турчин В.Ф. Эквивалентные преобразования программ на РЕФАЛе // Автоматизированная система управления строительством. - М.: ЦНИПИАС, 1974. - С. 36-68.
- Тыугу, 1984 Тыугу Э.Х. Концептуальное программирование. - М.: Наука, 1984. - 256 с.
- Тыугу и др., 1985 Тыугу Э.Х., Мацкин М.Б., Пеням Я.Э., Зомойс П.В. Объектно-ориентированный язык программирования НУТ // Прикладная информатика. - 1985. - Вып. 2. - С. 45-67.
- Уинстон, 1980 Уинстон П. Искусственный интеллект: Пер. с англ. - М.: Мир, 1980. - 519 с.
- Уинстон, 1987 Уинстон П. Лисп совершает революцию // Реальность и прогнозы искусственного интеллекта. - М.: Мир, 1987. - С. 71-84.
- Уолпер, 1981 Уолпер Л. Специальная машина, работающая на языке ЛИСП // Электроника. - 1981. - Т.54, N 17. - С. 4-6.
- Уолпер, 1982 Уолпер Л. Снижение цен на машины, работающие с языком ЛИСП // Электроника. - 1982. - Т. 55, N 17. - С.4-5.
- Фет, 1986 Фет Я.И. Вертикальная обработка как основа крупноблочной архитектуры // Изв. АН СССР. Техн. кибернетика. - 1986. - N 5. - С. 139-152.
- фон Нейман, 1971 фон Нейман Дж. Теория самовоспроизводящихся автоматов. - М.: Мир, 1971.
- Форсайт, 1987 Экспертные системы. Принципы работы и примеры // Под ред. Ф.Форсайт. - М.: Радио и связь. - 1987. - 224 с.
- Фостер, 1981 Фостер К. Ассоциативные параллельные процессоры. - М.: Энергоиздат, 1981. - 240 с.
- Фридман и др., 1978 Фридман А., Менон П. Теория проектирования перекрывающихся схем. - М.: Мир, 1978. - 500 с.
- Хартман, 1986 Хартман А.С. Разработчик перед выбором: программа или НС // ТИИЭР. - 1986. - Т.74, N 6.
- Хендерсон, 1982 Хендерсон П. Функциональное программирование. - М.: Мир, 1982. - 230 с.
- Хейес-Рот и др., 1987 Хейес-Рот Ф. и др. Построение экспертных систем. - М.: Мир, 1987. - 441 с.
- Хиллис, 1987 Хиллис У.Д. Коммуникационная машина // В мире науки. - М.: Мир, 1987. - N 8.
- Хорошевский, 1979 Хорошевский В.Ф. ATNL - язык представления лингвистических знаний в естественных языковых системах // Вопросы кибернетики. - 1979. - Вып. 55.

- Хорошевский, 1984  
Хорошевский В.Ф. Инструментальные экспертные системы // Представление знаний в человеко-машинных и робототехнических системах. Т.С. Прикладные человеко-машинные системы, ориентированные на знания / ВИНТИ. - М., 1984.- С. 329-367.
- Хорошевский, 1986  
Хорошевский В.Ф. Разработка и реализация экспертных систем -инструментальный подход // Изв. АН СССР. Техн. кибернетика. - 1986. - N 5. - С. 105-114.
- Щенников, 1987  
Щенников С.Ю. Методы и средства программирования механизмов управления знанием в экспертных системах // Проблемы разработки и внедрения экспертных систем в непромышленной сфере: Сб. науч. трудов по материалам школы. - Минск, 1988. - С. 94-102.
- Эйсымонт, 1977  
Эйсымонт Л.К. О возможности параллельных схем реализации одного языка для описания задач переработки текстовой информации // УСиМ, 1977. - N 2.- С. 56-64.
- Эйсымонт и др., 1982  
Эйсымонт Л.К., Платонова Л.Н. Выбор и оценка базового языка символьного процессора // Совещание по системам и методам аналитических вычислений на ЭВМ и их применению в теоретической физике. - Дубна, 1983. - С 19-33.
- Электроника, 1986  
Электроника. - 1986.- Т. 59, N 7,9, 12,16,18,22.
- Электроника, 1987а  
Электроника. - 1987. - Т. 60, N 5,6,8.
- Электроника, 1987б  
Почему фирма SUN решила разработать собственный RISC-микропроцессор // Электроника.- 1987. - Т.60, N 18. - С. 28-30.
- Abe et al., 1987  
Abe S. et al. High Performance Integrated Prolog Processor IPP // Computer Archit. News.-1987.- Vol. 15, N 2.- P. 100-107.
- Ackerman, 1982  
Ackerman W. Data Flow Languages// Computer.- 1982.-Vol. 15, N 2.- P. 15-25.
- Ackerman et al., 1979  
Ackerman G.B., Dennis J.B. VAL - a Value Oriented Algorithmic Language // Preliminary Reference Manual/Techn. Rep. NTR-218.Computation Structures Group, Lab. Computer Science. - MIT. - 1979, May, June.- 160 p.
- Adams et al., 1987  
Adams I.G., Agrawal D.P., Seigel H.J. A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks//Computer.- 1987.-N 1.
- Agerwala et al., 1982  
Agerwala T., Arvind D. Data Flow Systems // Computer.- 1982.- Vol. 15, N 2.- P. 10-13.
- Agrawal, 1982  
Agrawal D.P. Testing and Fault Tolerance of Multistage interconnection networks // Computer.- 1982, - Vol. 15, N 4.
- Agrawal, 1983  
Agrawal D.P. Graph Theoretical Analysis and Design of Multistage Interconnection Networks // IEEE Trans. - 1983.- Vol. Com - 17, N 7.

- Aiello et al.,  
1981  
Aiello L., Prini G. An Efficient Interpreter for the LAMBDA-calculus // J. Computer and Syst. Sci.-1981.- Vol. 23, N 3.- P. 383-424.
- ALSCA,  
1986  
Annual 13th International Symposium Computer Architecture. (ALSCA). Tokyo, Japan, 1986.
- Allan,  
1986  
Allan R. Technology Shoots for an Automated Battle Line // Electronic Des. - 1986. - Vol. 34, N 22. - P. 86-102.
- Allen,  
1978  
Allen J. Anatomy of LISP. - New-York: McGraw-Hill, 1978. - 273 p.
- Amundsen et al.,  
1985  
Amundsen M.J. et al. Compact Lisp Machine. // Proc. AIAA/ACM/NASA/IEEE Computer Aerospace Conf.- 1985.- P.422-426.
- Anderson,  
1986  
Anderson D.B. Experience with Flamingo: A Distributed Object-Oriented Interface System // ACM: Special Issue SIGPLAN Notice. - 1986. - Vol. 21, N 11. - P. 177-185
- Andre et al.,  
1980  
Andre F. et al. KENSUR - an Architecture, Oriented Towards Programming Language Translations. // SIGART Newsletter. - 1980. - Vol. 8, N 3. - P. 17-22.
- Anjewierden,  
1987  
Anjewierden A. Knowledge Acquisition Tool // AI Communications.- 1987.- Vol. 10, N 1. - P.29-38.
- Annaratone et al.,  
1986  
Annaratone et al. WARP Architecture and Implementation // Comp. Archit. News. - 1986. - Vol.14, N 2.- P. 346-357.
- Arrando et al,  
1985  
Arrando G. et al. Modeling Knowledge for Software Development // IEEE 3-rd Int. Workshop Software Specifications and Design. - 1985.
- ART,  
1984  
ART User's Manual. - Ca.: Inference Systems Inc., 1984. - 63p.
- Arvind et al.,  
1978  
Arvind D., Gostelow K.D., Pluffe W. The Preliminary Id Report // Techn. Rep. TR 114a, Dep. Information and Computer Science Univ. California Irvine. - 1978, May.- 180 p.
- Arvind et al.,  
1980  
Arvind D, Kathail V., Pingalli K.A. Data Flow Architecture with Tagged Tokens // MIT/LCS.- 1980, - TM-174, Sept.
- Atallah et al.,  
1985  
Atallah M.J., Kosaraju S.R. A Generalites Dictionary Machine for VLSI // IEEE Trans. - 1985.- N 2.
- Atkinson et al.,  
1987  
Atkinson R.R., McCreight E.M. The Dragon Processor // SIGPLAN Notices. - 1987. - Vol.22, 10. - P. 65-69.
- Backus,  
1987  
Backus J. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs // XCACM. - 1978. - Vol. 21, N 21. - P. 613-641.
- Baker,  
1978a  
Baker H.G. List Processing in Realtime on a Serial Computer // CACM.- 1978.- Vol.21, N 4.- P.280-293.
- Baker,  
1978b  
Baker H.G. Shallow Binding in LISP 1.5//CACM.- 1978.- Vol. 21, N 7.- P. 565-569.
- Banerjee et al.,  
1986  
Banerjee P., Kuo S.-Y., Fuchs W. K. Reconfigurable Cube-Connected Cycles Architectures // Test Conference. - 1986.

- Barstow,  
1984 Interactive Programming Environments / Ed. by Barstow D.R. et al. - New York: McGraw-Hill, 1984. -609 p.
- Barton et al.,  
1980 Barton R.R. et al. Overview and Status of Dorado Lisp // Conf. Rec. LISP Conference ( Stanford, Aug. 25-27, 1980). - LISP Company. - P. 243-247.
- Basili et al.,  
1984 Basili V.R., Perricone B.T. Software Errors and Complexity: An Empirical Investigation // CACM.- 1984. - Vol.27, N 1. - P. 42-52.
- Bate,  
1987 Bate R. Lisp Chips Team up With Supercomputers // Computer Des. - 1987.- Vol. 26, N 6.- P. 63.
- Beck et al.,  
1987 Beck B., Kasten B., Thakkar S. VLSI Assist for a Multiprocessor//SIGPLAN Notices.-1987.- Vol.22, N 10.-P. 10-20.
- Bell,  
1986 Bell C.G. RISC: Back to the Future?// Datamation.-1986.-Vol. 32, N 11.- P. 96-108.
- Ben-Bassat et al.,  
1980 Ben-Bassat N. et al. Pattern-Based Interactive Diagnosis of Multiple Disorders: The MEDAS System // Proc. IEEE Trans. Pattern Analysis and Machine Intelligence. - 1980. - Vol. 2, N 2. - P. 148-160.
- Berra et al.,  
1979 Berra B., Oliver E. The Role of Associative Array Processors in Data Base Machine Architecture// Computer.-1979.- Vol.12, N 3. -P. 53-61.
- Berra et al.,  
1987 Berra P.B., Troullinos N.B. Optical Techniques and Data/Knowledge Base Machines // Computer.-1987, Oct.- P. 59-70.
- Bhuyan et al.,  
1984 Bhuyan L.N., Agrawal D.P. Generalized Hypercube and Hyperbus Structures for a Computer Network // IEEE Trans. - 1984.- N 4.
- Bobrow,  
1975 Bobrow D.G. A Note on Hash Linking // CACM.- 1975. - Vol. 18, N 7. - P. 413-415.
- Bobrow et al.,  
1977 Bobrow D., Winograd T. An Overview of KRL, a Knowledge Representation Language // Cognitive Science. - 1977. - Vol.1, N 1. - P. 1-46.
- Bobrow et al.,  
1979a Bobrow D., Winograd T. KRL, Another Perspective // Cognitive Science. - 1979. - Vol. 3, N 1. - P.29-42.
- Bobrow et al.,  
1979b Bobrow D.G., Clark D.W. Compact Encoding of List Structure // ACM Trans. Prog. Lang. and Syst. - 1979. - N 1. - P. 266-286.
- Bobrow et al.,  
1986 Bobrow D.G. et al. CommonLoops: Merging Lisp and Object-Oriented Programming // ACM Special Issue SIGPLAN Notice. - 1986. - Vol.21, N 11. - P. 17-29.
- Bolc,  
1983 Bolc L. He Design of Interpreters, Compilers end Editors of Augmented Transition Networks. - New-York: Springer-Verlag. - 1983. - 214 p.
- Bochman,  
1982 Bochman G.V. Hardware Specification with Temporal Logic: an Example // IEEE Trans. - 1982.- Vol. C-31, N 3.- P. 223- 231.
- Boley,  
1980 Boley H. The Preliminary Servey of Artificial Intelligence Machines // SIGARTH Newsletter. - 1980.- N 72.- P.21-28.

- Bond, 1984** Bond J. Single-user Symbolic Processor Cuts AI System Costs//Computer Des.- 1984, Nov.- P.45-47.
- Bond, 1987** Bond J. Parallel-Processing Concepts Finally Come Together in Real Systems // Computer Des. - 1987. - Vol.26, N 11.- P. 51-74.
- Booch, 1986** Booch J. Object-Oriented Development // IEEE Trans. - 1986. - Vol. SE-12, N 2. - P. 211-231.
- Boral et al., 1983** Boral H., DeWitt D.J. Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines // Database Machines / Ed. by H.O. Leilich and M. Missikoff. - Berlin: Springer-Verlag, 1983. - P. 166-187.
- Borriello et al., 1987** Borriello G. et al. RISCs vs CISCs for Prolog:A Case Study // SIGPLAN Notices. - 1987. - Vol.22, N 10. - P. 136-145.
- Borland, 1985** Turbo Pascal Reference Guide, Borland Int. - 1985. - 313 p.
- Borland, 1986** Turbo Prolog Reference Guide, Borland Int. - 1986. - 152 p.
- Borland, 1987** Turbo C Reference Guide, Borland Int., - 1987. - 535p.
- Bowen, 1983** Bowen D.L., Byrd L.M., Clocksin W.F. A Portable Prolog Compiler // Logic Programming Workshop'83. - Universidade Nova de Lisboa, 1983. - P. 74-83.
- Brachman, 1979** Brachman R. On the Epistemological Status of Semantic Networks // Associative Networks. - New York: Academic Press, 1979. - P. 27-56.
- Bolc, 1983** Bolc L. The Design of Interpreters, Compilers and Editors of Augmented Transition Networks. - New-York: Springer-Verlag. - 1983. - 214 p.
- Brachman et al., 1981** Brachman R.J., Israel J.J. A Report on KL-ONE // Annual Report "Research in Knowledge Representation for Natural Language Understanding", 1.9.1980-31.8.1981, BBN Report N 4785, p. 49-178.
- Brownston et al., 1985** Brownston L. et al. Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming // Addison-Wesley Publ. Co, Inc., 1985. - 457 p.
- Bruynooghe, 1982** Bruynooghe M., A Note on Garbage Collection in Prolog Interpreters // Proc. First Int. Logic Programming Conf. - University Marseille, 1982, Sept. - P. 52-55.
- Bryant, 1980** Bryant R.E. Report of the Workshop on Selftimed Systems//MIT Lab. Comput. Sci. Techn. Memo.- Cambridge, 1980.- N 166.- 21 p.
- Burks, 1960** Burks A.W. Computation, Behavior and Structure in Fixed and Growing Automata // Self-organizing Systems. - New York, 1960.
- Bush, 1945** Bush V. As We May Think//Atlantic Monthly. 1945. - Vol. 176, Jul. - 101 p.
- CARNEGIE, 1987** CARNEGIE GROUP INC. Expert Systems.-1987.- Vol.4, N 2, - 105 p.

- Castan et al.,  
1982 Castan M., Organick E.I. m3L: an HLL-RISC Processor for Parallel Execution of RP-language Programs // SIGPLAN Notices. - 1982.- P.239-247.
- Catier,  
1987 Catier E. Les Problemes des Machines LISP // Electronique Industrielle. - 1987, May.- N 125/1.- P. 65-67.
- Cavarroc et al.,  
1974 Cavarroc J.C., Blanshard M., Gillon J. An Approach to the Modular Design of Industrial Switching Systems // Proc. Int. Symp. Discrete Systems. - 1974. - Vol. 3. - Riga. - P. 93-102.
- Champine,  
1978 Champine G.A. Four Approaches to Data Base Computer // Datamation.- 1978.- Vol. 24, N 13.- P. 101-106.
- Charniak et al.,  
1980 Charniak E., Riesbeck C., McDermott D. Artificial Intelligence Programming. - New Jersey: Lawrence Erlbaum Associates, 1980. - 352p.
- Chow et al.,  
1987 Chow P., Horowitz M. Achitectoral Tradeoffs in the Design of MIPS-X//Comp. Archit. News.- 1987.-Vol. 15, N 2.- P. 300-308.
- Clark et al.,  
1977 Clark D.W., Green C.C. An Empirical Study of List Structure in Lisp // CACM.- 1977.- Vol. 20, N 2.- P. 78-87.
- Clark,  
1979 Clark D.W. Measurments of Dynamic List Structure Use in Lisp // IEEE Trans. - 1979.- Vol. SE-5, N 1.- P. 51-59.
- Clark et al.,  
1984 Clark K., Gregory S. Parlog: Parallel Programming in Logic // Doc Research Report 84/4. Imperial College Science and Technology, Department Computing, 1984. - 186 p.
- Clocksins,  
1981 Clocksin W. F., Mellish C. S. Programming in Prolog. - Springer-Verlag. - 1981. - 173p.
- Codd,  
1970 Codd E.A. A Relational Model of Data for Large Shared Data Banks//CACM.- 1970.- Vol.13, N 6.
- Colmerauer et al.,  
1981 Colmerauer A., Kanou H., Van Canghem M. Last Steps Towards an Ultimate Prolog // IJCAI-B1. - 1981.- P. 947-948.
- Colmerauer,  
1983 Colmerauer A. PROLOG in 10 Figures // Proc. Int. Joint Conf. AI. - 1983. - P.488-499.
- Colwell et al.,  
1987 Colwell R.P. et al. A VLIW Architecture for a Trace Scheduling Compiler //SIGPLAN Notices. - 1987. - Vol. 22, N 10.- P. 180-192.
- Commoner et al.,  
1971 Commoner F. et al. Marked Directed Graphs//J. Comput. and Syst. Sci.- 1971.- N 5.- P. 511-523.
- Computer,  
1979 Computer. -1979.- Vol. 12, N 3 (Спецвыпуск по машинам баз данных).
- Computer Des.,  
1987 AI Workstation Built Around Lispchip // Computer Des. - 1987.- Vol.26, N 14.- 110 p.
- Computerworld,  
1986 Floating Point's Hypercube Architecture Aimed Beyond Cray// Computerworld.-1986.-Vol.20, N 4.
- Conery et al.,  
1981 Conery J.S., Kibler D.F. Parallel Interpretation of Logic Programs // Proc. Conf. on Functional Programming Languages and Computer Architectures, - 1981. - P. 167-170.

- Conway,  
1963 Conway M. E. Design of a Separable Transition Diagram Compiler // CACM. - 1963. - Vol. 6, N 7. - P. 396-408.
- Corley et al.,  
1985 Corley C.J., Stats J.A. LISP Workstation Brings AI Power to a User's Desk // Computer Des. - 1985, Jan.-P. 155-162.
- Curry,  
1984 Curry J. Language-based Architecture Eases System Design // Computer Des. - 1983, Jan. - P.127-136.
- Database Eng.,  
1981 IEEE Database Engineering Bulletin - 1981. - Vol.4,N2 (Спецвыпуск по машинам баз данных).
- Database Eng.,  
1982 IEEE Database Engineering Bulletin -1982.- Vol. 5, N 3 (Спецвыпуск по оптимизации запросов).
- Database Eng.,  
1985 IEEE Database Engineering Bulletin -1985.- Vol. 8,N1 (Спецвыпуск по оценке производительности систем управления базами данных).
- Davies et al,  
1973 Davies D. et al. POPLER 1.5 Reference Manual. - Edinburg: University Edinburg, 1973. - 192p.
- Davis et al.,  
1982 Davis A.L, Keller R.M. Data-Flow Program Graphs//Computer.- 1982.- Vol.15, N 2.-P. 26-41.
- Davis et al.,  
1985 Davis N.J., Hsu W.T.-Y., Siegel H.J. Fault Location Techniques for Distributed Control Interconnection Networks // IEEE Trans. - 1985.- N 10.
- Dennis,  
1970 Dennis J.B. Modular Asynchronous Control Structures for High Performance Computer // Rec. Project MAC Cong. on Concurrent Systems and Parallel Computation, ACM. - N.Y., 1970. - P. 55-80.
- Derksen,  
1973 Derksen J. A. C. QA-4: A Language for Artificial Intelligence. - California: SRI, Menlo Park, 1973.
- Despain et al.,  
1984 Despain A.M., Patt Y.N. The Aquarius Project // Digest of Papers. - Compcon.- IEEE Press.- Spring 1984.- P. 364-367.
- Deutsch,  
1978 Deutsch L.P. Experience With a Microprogrammed Interlisp System // Proc. MICRO-11. Asilomar, Nov.).- (IEEE),ACM. - 1978.- P.128-129.
- Deutsch,  
1980 Deutsch L.P. ByteLisp and its Alto Implementation // Conf. Rec. 1980 LISP Conf. (Stanford, Aug. 25-27). - LISP Company. - 1980.- P. 231-242.
- DeWitt,  
1979 DeWitt D. DIRECT - A Multiprocessor Organization for Supporting Relational Data Base Management Systems // IEEE Trans. - 1979.-Vol. C-28.- P. 395-406.
- Dios et al.,  
1981 Dios D.M., Jump J.R. Analysis and Simulation of Buffered Delta Networks // IEEE Trans. - 1981. - N 4.
- Ditzel et al.,  
1987 Ditzel D.R., McLellan H.R., Berenbaum A.D. The Hardware Architecture of the CRISP Microprocessor // Comp. Archit. News.- 1987.- Vol.15, N 2.- P. 309-319.
- Dobry et al,  
1984 Dobry T.P., Patt V.N., Despain A.M. Design Decision Influencing the Microarchitecture For a Prolog Machine // MICRO 17, Proc.1984.

- Dobry,  
1985 Dobry T.P., Despain A.M., Patt V.N. Performance Studies of a Prolog Machine Architecture // Proc. 18-th Ann. Workshop Microprogramming. - 1985.
- Dobry et al.,  
1985 Dobry T.P., Despain A.M., Patt V.N. Performance Studies of a Prolog Machine Architecture // SIGARCH Newsletter. -1985. - Vol. 13, N 3. - P. 180-190.
- Dubois et al.,  
1988 Dubois M., Scheurich C., Briggs F. Synchronization, Coherence and Event Ordering in Multiprocessors. // Computer. - 1988, Feb.- P. 9-21.
- EXSYS,1985 EXSYS Reference Manual. - EXSYS Inc., 1985.
- Ewing,  
1986 Ewing J.J. An Object-Oriented Operating System Interface // SIGPLAN Notices. - 1986. - Vol. 21, N 11. - P. 30-37.
- Fagin,et al.,  
1985 Fagin B., Patt Y., Srimi V., Despain A. Compiling Prolog Into Microcode: A Case Study Using the NCR/32-000//Proc.18th Ann. Workshop Microprocessors.-1985.- P.79-88.
- Fagin et al.,  
1987 Fagin B.S., Despain A.M. Performance Studies of a Parallel PROLOG Architecture//Computer Archit. News.-1987.- Vol.15, N 2.- P. 108-116.
- Fahlman et al.,  
1983 Fahlman S.E. et al. Massively Parallel Architectures for AI: NETL, Thistle and Boltzmann Machines // Proc. Nat. Conf. Art. Int. - Washington, D.C., 1983, Aug.- P. 109-113.
- Fateman,  
1978 Fateman R. J. Is Lisp Machine Different from Fortran Machine? // SIGSAM Bull.- 1978.- Vol. 12, N 3.- P. 8-11.
- Feng et al.,  
1981 Feng T.-Y., Wu C.-L. Fault-diagnosis for a Class of Multistage Interconnection Networks // IEEE Trans. Comput.- 1981.- N 10.
- Fenichel et al.,  
1969 Fenichel R.R., Yochelson J.C. A LISP Garbage Collector for Virtual Memory Computer Systems // CACM.-1969.- Vol.12, N 11.- P. 611-612.
- Fennell et al.,  
1977 Fennell R.D., Lesser V.R. Parallelism in Artificial Intelligence Problem Solving: a Case Study of Hearsay II // IEEE Trans. - 1977.- N 2. Proc. Conf. Fifth Generation Computer Systems, (FGCS). - 1984, Nov. 6-9, Tokyo, Jap., ICOT.
- FGCS,  
1984 Fisher J.A. Very Long Instruction Word Architectures and the ELI-512 // SIGARCH Newsletter. - 1983. - Vol.11, N 3. - P. 140-150.
- Fisher,  
1983 Fisher A.L. Dictionary Machines with a Small Number of Processors//SIGARCH Newsletter.-1984.- Vol.12, N 3.-P. 151-156.
- Fisher,  
1984 Fitch J.P. Manual for Standard LISP on IBM System 360 & 370 // Tech. Rep. TR-6. Utah Symbolic Computation Group, 1978. - 93 p.
- Florentin,  
1987 Florentin J.J. Software Review // KEE, Expert Systems. - 1987. - Vol. 4, N 2. - P. 118-120.
- Foderaro et al.,  
1983 Foderaro J.K., Skowler K.L., Laver K. The FRANZLISP Manual. - Berkeley: University California, 1983. - 194 p.



- Fox et al.,  
1986 Fox E.R., Kiefer K.J., Vangen R.F., Whalen S.P. Reduced Instruction Set Architecture for GaAs Microprocessor System//Computer.-1986.- Vol. 19, N 10.- P. 71-81.
- Frankel,  
1986 Frankel K.A. Evaluating two Massively Parallel Machines // CACM. -1986.- Vol.29, N 8.- P. 752-758.
- Frenkel,  
1985 Frenkel K.A. Towards Automating the Software Development Cycle // CACM. - 1985. - Vol.28, N 6. - P. 578-589.
- Friedman et al.,  
1978a Friedman D.P., Wise D.S. Aspects of Applicative Programming for Parallel Processing // IEEE Trans. - 1978.- Vol. C-27, N 4.- P. 64-71.
- Friedman et al.,  
1978 Friedman D.P., Wise D.S. Aspects of Applicative Programming for Parallel Processing // IEEE Trans. - 1978.- Vol. C-27, N 4.- P. 289-296.
- Fukunada et al.,  
1986 Fukunada K., Hirose S. An Experience With a Prolog-based Object-Oriented Language//ACM. Special Issue SIGPLAN Notice. - 1986. - Vol. 21, N 11. - P. 224-231.
- Furht et al.,  
1987 Furht B., Milutinovic V. A Survey of Microprocessor Architectures for Memory Management // Computer.- 1987, Mar.- P. 48-67.
- Gabriel,  
1985 Gabriel R.P. Performance and Evaluation of Lisp System // MIT- Press.-1985.
- Gabriel et al.,  
1982 Gabriel R.P., Masinter L.M. Performance of Lisp Systems // Conf. Rec. ACM Symp. Lisp and Functional Programming.-1982.- P. 132-142.
- Gajski et al.,  
1984 Gajski D., Kim W., Fushimi S. A Parallel Pipelined Relational Query Processor: An Architectural Overview//SIGARCH Newsletter.- 1984. - Vol.12, N 3.- P. 134-141.
- Gajski et al.,  
1985 Gajski D.D., Peir J.-K. Essential Issues in Multiprocessor Systems //Computer.- 1985.- Vol.18, N 6.- P. 9-27.
- Gilbert et al.,  
1986 Gilbert B.K. et al. Signal Processors Based Upon GaAs ICs: The Need for a Wholistic Design Approach//Computer.- 1986.- Vol. 19, N 10.- P. 29-43.
- Gimarc et al.,  
1987 Gimarc C.E., Milutinovic V.M. A Survey of RISC Processors and Computers of the Mid-1980s//Computer.- 1987, Sept. - P. 59-69.
- Goldberg et al.,  
1983 Goldberg A., Robson D. Smalltalk-80. The Language and its Implementation. - Addison Wesley, 1983. - 714 p.
- Golden,  
1985 Golden Common LISP. - Gold Hill Computers Co Manual, 1985. - 379p.
- Goldstein et al.,  
1977 Goldstein I., Bruce R. NUDGE - a Knowledge-based Scheduling Program // Proc. IJCAI - 1977. - P. 257-283.
- Goncalves et al.,  
1988 Goncalves G. et al. A Network of Transputers to Emulate a Parallel Symbolic Processor // Microprocessing and Microprogramming.- 1988.- Vol. 23.- P. 149-152.

- Goodman, 1987 Goodman J.R. Coherency for Multiprocessor Virtual Address Caches // SIGPLAN Notices. - 1987. - Vol. 22, N 10.- P. 72-81.
- Gordon et al., 1984 Gordon D., Koren I., Silberman G.M. Embedding Tree Structures in VLSI Hexagonal Arrays // IEEE Trans. - 1984. - N 1.
- Goto et al., 1984 Goto A., Tanaka H., Moto-Oca T. Highly Parallel Inference Engine PIE-Goal Rewriting Model and Machine Architecture // New Generation Computing. - 1984. - Vol. 2, N 1.- P. 37-58.
- Greeger, 1983 Greeger M. Lisp Machines Come out of the Lab. // Computer Des. - 1983, Nov. - P. 207-216.
- Green, 1969 Green C. Theorem Proving by Resolution as a Basis for Question Answering Systems // Machine Intelligence / Edinburg Univ. Press. - 1969. - Vol. 4. - P. 183-205.
- Greenfeld, 1981 Greenfeld N.R. Jerico-a Professional's Personal Computer System // SIGART Newsletter. - 1981. - Vol.9, N 3.- P. 217-226.
- Greiner et al., 1980 Greiner R., Lenat D., A Representation Language // Proc. Int. Conf. AI. - 1980. - P. 165-169.
- Griss et al., 1977 Griss M.L., Swanson M.R. MBALM/1700: A Microprogrammed Lisp Machine for the Burroughs B1726 // Proc. MICRO-10 ACM.-New York, 1977.- P. 15-25.
- Griss et al., 1978 Griss M., Kessler R. REDUCE/1700: a Microcoded Algebra System // Proc. MICRO-11 (IEEE). - 1978. - P. 130-138.
- Griss et al., 1981 Griss M.L., Hearn A.C. A Portable LISP Compiler // Software - Practice and Exper.- 1981.- 11.- P. 541-605.
- Griswold, 1978 Griswold R.EJ. A History of the SNOBOL Programming Language// SIGPLAN Notices.- 1978. -Vol.13, N 8.P. 275-308.
- Guha et al., 1987 Guha A., Ramnarayan R. Architectural Issues in Designing Symbolic Processors in Optics // Comp. Archit. News. - 1987. - Vol. 15, N 2. - P. 145-151.
- Gupta et al., 1987 Gupta A., Forgy C., Newell A., Wedig R. Parallel Algorithms and Architectures for Rule-Based Systems // Comp. Archit. News.- 1987.- Vol. 14, N 2.- P. 28-37.
- Gurd et al., 1983 Gurd J.R., Watson I. Preliminary Evaluation of a Prototype Dataflow Computer // Proc. IFIP. - 1983. - P. 245-551.
- Gurd, 1985 Gurd J.R. The Manchester Dataflow Machine // Comput. Phys. Commun.-1985. - Vol.37, N 1-3.
- Gurd et al., 1985 Gurd J.R., Kirkham C.C., Watson I. The Manchester Prototype Dataflow Computer//CACM.- 1985. - Vol. 28, N 1.- P. 34-52.
- Gusman, 1981 Gusman A. A Parallel Heterarchical Machine for High Level Language Processing // Proc. Int. Conf. Parallel Process. - New York, 1981.- P. 64-71.

- Guy et al.,  
1984  
Haber,  
1986  
Haberman et al.,  
1986  
Halstead,  
1984  
Halstead,  
1985  
Halstead,  
1986  
Halstead et al.,  
1986  
Hannessy et al.,  
1982  
Hannessy et al.,  
1983  
Hannessy,  
1984  
Hansen,  
1969  
Haridi,et.al.,  
1984  
Harmon,  
1987  
Hasegawa et al.,  
1986  
Hayashi et al.,  
1983  
Hayashi et al.,  
1984
- Guy L., Steele R. Common Lisp: The Language. Digital Press, 1984. - 241p.  
Haber L. Xerox Proves it Moves Ideas Into the Market // Mini-micro Systems.- 1986, March.- P. 33-40.  
Haberman A.N., Notkin D. Gandalf: Software Development Environments // IEEE Trans. - 1986. - Vol. SE-12, N 12. - P. 1117-1127.  
Halstead R.H., Jr. Implementation of MultiLisp: Lisp on a Multiprocessor//Conf. Rec. ACM Symp. Lisp and Functional Programming.- 1984.- P. 9-17.  
Halstead R.H., Jr. Multilisp: A Language for Concurrent Symbolic Computation//ACM Trans. Progr. Lang. and Systems.- 1985.- Vol. 7, N 4.- P. 501-538.  
Halstead R.H., Jr. Parallel Symbolic Computing//Computer.- 1986.- Vol. 19, N 8.- P. 35-43.  
Halstead R.H., Jr., Anderson T.L., Osborne R.B., Sterling T.L. Concept: Design of a Multiprocessor Development System//Comp. Achit. News.- 1986.- Vol. 14, N 2.- P. 40-48.  
Hannessy J.L., Gross T.R. Code Generation and Reorganization in the Presence of Pipeline Constraints // Conf. Rec. 9th ACM Symp. Princ. Progr. Lang., Albuquerque. - New York, N.Y.- 1982.- P. 120-127.  
Hannessy J.L., Gross T. Postpass Code Optimization of Pipeline Constraints // ACM Trans. Progr. Lang. and Systems.- 1983.- Vol.5, N 3.- P. 422-428.  
Hannessy J.L. VLSI Processor Architecture//IEEE Trans. - 1984.- Vol. C-33, N 12.- P. 1221-1246.  
Hansen W.J. Compact List Representation: Definition, Garbage Collection and System Implementation // CACM.- 1969.- Vol. 12, N 9.- P.499-507.  
Haridi S., Sahlin D. Efficient Implementation of Cyclik Structures // Implimentation Prolog / Ed. by J.A. Campbell. - Ellis Horwood, 1984. - p. 114-178.  
Harmon P. Tools // Expert Systems Strategies. - 1987. - Vol. 3, N 6. - P. 11-18.  
Hasegawa M., Shigei Y. AT-O(N log N), T-O(log N) Fast Fourier Transform in a Light Connected 3-Dimensional VLSI // Comp. Archit. News.- 1986.- Vol. 14, N 2.- P. 252-260.  
Hayashi H., Hattori A., Akimoto H. ALPHA: A High-Performance LISP Machine Equipped With a New Stack Structure and Garbage Collection System // SIGARTH Newsletter. - 1983. - Vol.11, N 3. - P. 342-348.  
Hayashi H., Hattori A., Akimoto H. Lisp Machine "ALPHA"//Fujitsu Scientific and Technical. - 1984. - Vol. 20, N 2.- P. 219-234.

- Hayes-Roth, 1985  
 Hayes-Roth B., A Blackboard Architecture for Control // Artificial Intelligence, 1985. - Vol.26. - P. 251-321.
- Haynes et al., 1982  
 Haynes L.S., Lou R.L., Siewiorek D.P., Mizell D.W. A Survey of Highly Parallel Computing // Computer.- 1982, Jan.
- Hennessey, 1987  
 Hennessey J. RISK: the Performance Standard for Design // Computer Des.-1987.- Vol. 26, N 10.- P. 56.
- Hennessey et al., 1982  
 Hennessey J. et al. Hardware/Software Tradeoffs for Increased Performance//SIGPLAN Notices.- 1982.- Vol. 17, N 4.- P. 2-11.
- Hewitt, 1973  
 Hewitt C. An Universal Modular ACTOR Formalism for Artificial Intelligence // Proc. IJCAL. - California: Stanford, 1973. - P. 235-245.
- Hibino, 1980  
 Hibino Y. A Practical Parallel Garbage Collection Algorithm and its Implementation // SIGARTH Newsletter. -1980.- Vol.8, N 3.- P. 113-120.
- Highberger et al., 1984  
 Highberger D., Edson D. Intelligent Computing Era Takes Off//Computer Des. - 1984, Sept.- P.79-95.
- Hill et al., 1986  
 Hill M. et al. Design Decision in SPUR//Computer.-1986.-Vol.19. N 10,11.
- Hillis, 1985  
 Hillis D.W. The Connection Machine// MIT Press.- Cambridge, Mass., 1985.
- Hillyer et al., 1986  
 Hillyer B.K., Shaw D.E., Nigam A. NON-VON's Performance on Certain Database Benchmarks//IEEE Trans. -1986.- N 4.- P. 577-583.
- Hindin, 1984  
 Hindin H.J. Fifth-generation Computing: Dedicated Software is the Key // Computer Design.- 1984, Sept.- P. 150-164.
- Hopfield et al., 1986  
 Hopfield, Tank. Computing with Neural Circuits: an Model // Science. - 1986. - Vol. 233, Aug.
- Huany et al., 1986  
 Huany S.-T., Tripathi S.K. Finite State Model and Compatibility Theory: New Analysis Tools for Permutation // IEEE Trans. - 1986.- N 7.
- Huffman, 1954  
 Huffman D.A. The Synthesis of Sequential Switching Circuits // J. Franklin Inst., 1954.- Vol. 257, N 3,4.- P. 161-190, 275-303.
- Hull, 1987  
 Hull M.E.C. Occam - a Programming Language for Multiprocessor Systems // Comput. Lang.- 1987.- Vol. 12, N 1.- P. 27-37.
- Hwong et al., 1987  
 Hwong K., Ghosh J., Chowkwanyun R. Computer Architectures for Artificial Intelligence Processing//Computer.- 1987.- N 1.
- Ida et al., 1981  
 Ida T., Itano K. Associative Description Scheme for the Exploitation of Address Arithmetic in LISP // J.Inform. Process.- 1981.- N 3.- P. 147-151.
- Inmos, 1984  
 Inmos Limited // IMS T424 Transputer Reference Manual.- England, 1984.
- Ishiwkawa et al., 1986  
 Ishiwkawa Y., Tokoro M. A Concurrent Object-Oriented Knowledge Representation Language Orient84/k: Its Features and Implementation //

- SIGPLAN Notices. - 1986. - Vol. 21, N 11. - P. 232-241.
- Jackson, 1975 Jackson M. Principles of Program Design. - New York: Academic Press, 1975. -179 p.
- Jump et al., 1975 Jump J.R., Thiagarajan P.S. On the Interconnection of Asynchronous Control Structures // ACM. - 1975. - Vol. 22, N 4.- P. 102-112.
- Kakuta et al., 1985 Kakuta T. et al. The Design and Implementatuon of Relational Database Machine Delta//Proc. Int. Workshop Database Machines'85.- 1985, Mar.
- Kanamory et al., 1986 Kanamory T., Fujita H., Seki H., Horiuchi K., Maeji M. ARGUS/5:A System For Verification of Prolog Programs // Proc. ACM/IEEE Computer Society Fall Joint Computer Conf. - 1986. - P. 994-999.
- Kaneda et al., 1986 Kaneda Y., Tamura N., Wada K.,Matsuda H.,Kuo S., Maekawa S. Sequential Prolog Machine PEK // New Generation Computing.- 1986.-N 4.-P.51-66.
- Kaplan, 1987 Kaplan I. The LDF100: A Large Grain Dataflow Parallel Processor // Computer Archit. News.-1987.-Vol.15, N 3.- P. 5-12.
- Karp et al., 1986 Karp S., Roosild S. DARPA, SDI, and GaAs//Computer.-1986.- Vol. 19, N 10.- P. 17-19.
- Kelem, 1985 Kelem S.H. A Method for the Automatic Translation of Algorithms from a High-level Language Into Self-timed Integrated Circuits // IEEE Circuits and Devices Mag., 1985. - Vol.1, N 2.- P. 17-19, 44.
- Keller, 1980 Keller R.M. Divide and CONCer: Data Structuring in Applicative Multiprocessing Systems // Conf. Rec. ACM Lisp Conf. - 1980. - P.196-202.
- Kerlis et al., 1985 Kerlis P.A. et al. The Saga Approach to Large Program Development in an Intgrated Modular Environment// Proc. GTE Workshop on Software Engineering Environments for Programming-in-Large (June, 1985).
- Khoroshevsky, 1983 Khoroshevsky V.F. ATNL-machine - Software & Hardware // Proc. 1st Symp. IFAC on AI - USSR, 1983. - 12p.
- Khoroshevsky et al., 1987 Khoroshevsky V.F., Sherstnev V.Yu. REBUS - Knowledge Representation Tools for Expert Systems // Proc. Artificial Intelligence and Information-Control Systems Robots-87 Conf. / Ed. by I.Plander. - Elsevier Science Publishers B.V. (North-Holland), 1987. - p. 287-291
- Kleer, 1986 de Kleer J. Assumption-based Truth Maintenance Strategy // Artifitial Intelligence. - 1986. - Vol. 28, N 2. - P. 263-275.
- Kohli et al., 1987. Kohli M., Giuliano M.E., Minker J. An Overview of the PRISM Project //Computer Archit. News.- 1987.- Vol. 15, N 1.- P. 35-42.
- Komorowski, 1983 Komorowski H.J. An Abstract Prolog Machine // Integrated Computing Systems Proc./ Ed. by P. Degano, E. Sandewall. - North-Holland Publ. Co., 1983. - 76p.

- Kondo et al.,  
1986  
Kondo T. et al. Pseudo MIMD Array Processor-AAP2 // Computer Archit. News.- 1986.- Vol.14, N 2.- P. 330-337.
- Koster,  
1980  
Koster A. An Algorithm for Translating LIS<sup>2</sup> Programs into Reduction Language Programs // Lect. Notes Comput. Science. -1980.- N 83.
- Kowalski,  
1979  
Kowalski R. Algorithm = Logic + Control//CACM,- 1979.-Vol.22, N 7. - p. 424-436.
- Kung,  
1982  
Kung H.T. Why Systolic Architecture? //Computer.- 1982.- Vol.15, N 1. - P.37-46.
- Kung,  
1984  
Kung H.T. Systolic Algorithms // Large Scale Scientific Computation. - 1984. - Vol.51. - P. 127-139.
- Lampson et al.,  
1980  
Lampson B.W., Kenneth A.P. A Processor for High Performance Personal Computer // SIGARCH Newsletter.- 1980.- Vol. 8, N 3.- P. 146-160.
- Leavenworth,  
1966  
Leavenworth B.M. Syntax Macros on Extended Translation // CACM. - 1966. - Vol. 9, N 10. - P. 790-793.
- Lecht,  
1977  
Lecht C.P. The Waves of Change // Computerworld.- 1977.- Vol.11, N 27.- P.11-14.
- Lehnert et al.,  
1979  
Lehnert W., Wilks Y. A Critical Perspective of KRL // Cognitive Science. - 1979.-Vol. 3, N 1. - P. 1-28.
- Lehr et al.,  
1987  
Lehr T.F., Wedig R.G. Toward a GaAs Realization of a Production-System Machine // Computer.- 1987, Apr.- P. 36-48.
- Lenat,  
1983  
Lenat D. EURISKO: A Program, that Learns New Heuristics and Domain Concepts // Artificial Intelligence. - 1983. - Vol. 21. - P. 61-98.
- Lewis et al.,  
1979  
Lewis G.R., Henry J.S. The BTI-8000-Homogeneous, General Purpose Multiprocessor //Proc. AFIPS Conf.- 1979.- P. 513-528.
- Li et al.,  
1986  
Li K., Hudak P. New List Compaction Method // Software - Practice and Experience.-1986.- Vol. 16, N 2.- P.145-163.
- Lieberman et al.,  
1983  
Lieberman H., Hewitt C. A Real-Time Garbage Collector Based on the Lifetimes of Objects // CACM.- 1983.- Vol. 26, N 6.- P. 419-429.
- Liskov et al.,  
1974  
Liskov B., Zilles S. Programming with Abstract Data Types // SIGPLAN Notices.- 1974.- Vol.9, N 4.- P.50-59.
- Liskov,  
1976  
Liskov B. An Introduction to CLU // New Directions in Algorithmic Languages / Ed. by S.A. Schuman. - IRIA. - 1976. - P. 139-156.
- Lloyd,  
1984  
Lloyd J. W., Foundation of logic Programming.- New-York: Springer-Verlag. - 1984. - 215p.
- Lozovsky,  
1982  
Lozovsky V.S. An Integrated Knowledge Representation System for Situational Management // Computers and Artificial Intelligence. - 1982. - Vol. 1, N 4. - P. 317-329.
- Maier et al.,  
1984  
Maier D., Copeland G. Making Smalltalk a Database System // ACM SIGMOD Rec. -1984. - Vol.14, N 2. - P. 316-324.

- Maier et al.,  
1986  
Maier D., Stein J., Otis A., Purdy A. Development of an Object-Oriented DBMS// SIGPLAN Notices. - 1986. - Vol. 21, N 11. - P. 347-349.
- Manna,  
1975  
Manna Z., Waldinger R. Knowledge and Reasoning in Program Synthesis // Artificial Intelligence. - 1975. - N 6. - P. 175-208.
- Malachi et al.,  
1981  
Malachi Y., Owicki S.S. Temporal Specification of Self-timed Systems // Proc. VLSI Syst. and Comput. Carnegie-Mellon Univ. Conf.- Pittsburgh, Pa.- 1981.- P. 203-212.
- Mamdani,  
1977  
Mamdani E.H. Application of Fuzzy Sets Theory to Control Systems: a Survey // Fuzzy Automata and Decision Processes Ed.by M.M.Gupta, G.Saridis, B. Gaines. -Amsterdam: North-Holland, 1977.
- Marti et al.,  
1979  
Marti J.B., Hearn A.C., Griss M.L., Griss C. Standard LISP Report //ACM SIGPLAN Notices. - 1979. - Vol. 14, N 10. - P. 48-68.
- Marti et al.,  
1985  
Marti J.B., Hearn A.C. REDUCE as a LISP Benchmark // SIGSAM Bulletin. - 1985. -Vol.19, N 3.- P. 8-16.
- Martin,  
1986  
Martin A.J. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits // Distributed Computing. 1986.- Vol.1, N 4.- P. 205-225.
- Masinter et al.,  
1985  
Masinter L., van Melle W. Measuring Lisp Performance//IEEE SPRING COMCON. -1985. -P. 402-404.
- Matthews et al.,  
1986  
Matthews G., Manual G., Krueger S. Matching Hardware to Lisp Yields Peak Performance // Computer Des.- 1986. - Vol. 25, N 9.- P. 95-98.
- Matthews et al.,  
1987  
Matthews G., Hewes R., Krueger S. Single-chip Processor Runs LISP Environments // Computer Des. - 1987, May. - P. 69-76.
- Mattos,  
1984  
Mattos P. The Transputer//New Electronics.- 1984.- Vol. 17, N 16.- P. 43-45.
- McDermott,  
1972  
McDermott D. N., Susman G. J. The CONNIVER Reference Manual.- AI MEMO 259, AI Lab.,MIT, Mass., 1972.
- MDBS,  
1986  
MDBS Comp. - Guru Reference Guide. - 1986.
- McDonald et al.,  
1987  
McDonald J.E. et al. Wafer Scale Interconnections for GaAs Packaging - Applications to RISC Architecture // Computer. - 1987. - apr.- P. 21-35.
- McCarthy,  
1978  
McCarthy J. History of LISP// SIGPLAN Notices.- 1978.-Vol.13.-P.217-223.
- McNeley et al.,  
1987  
McNeley K.J., Milutinovic V.M. Emulating a Complex Instruction Set Computer with a Reduced Instruction Set Computer // IEEE Micro.- 1987, Febr.- P. 60-72.
- Melenek et al.,  
1987  
Melenek H.,Neun W. Announcement of REDUCE 3.2 for Gray X-MP // SIGRAM Bulletin.-1987.- Vol.21, N 2.- P. 3.

- Melikhov et al.,  
1987 Melikhov A.N., Bershtein L.S., Korovin S.Ya. Making Control Decisions in Fuzzy Systems Invariant to a Change of External Conditions // Fuzzy Sets and Systems.- 1987. - N 22.
- Mellish,  
1982 Mellish C.S. An Alternative to Structursharing in the Implimentation of a PROLOG Interpreter // Logic Programming / Ed. by Clark K.L. and Toprulund. - Acaddemic Press, 1982.
- Meng,  
1986 Meng B. AI Wars:Garbage Collection Gets Serious // Digital Design.-1986.- Vol. 16, N 11.- P. 48-51.
- Meshach,  
1984 Meshach W. Data-flow IC Makes Short Work of Tough Processing Chores // Electronic Des.- 1984. - May.- P. 191-206.
- Mills,  
1987 Mills J.W. Coming to Grips with a RISC: A Report of the Progress of the LOW RISC Desing Group // Computer Archit. News.-1987.-Vol.15, N 1.- P. 53-62.
- Milutinovic et al.,  
1986a Milutinovic V., Fura D., Helbig W. An Introduction to GaAs Microprocessor Architecture for VLST//Computer.- 1986. - Mar.- P. 30-42.
- Milutinovic et al.,  
1986b Milutinovic V. et al. Issues of Importance in Designing GaAs Microcomputer Systems//Computer. - 1986.- Vol. 19, N 10.- P. 45-57.
- Milutinovic,  
1986c Milutinovic V. GaAs Microprocessor Technology // Computer.- 1986.- Vol. 19, N 10.- P. 10-13.
- Milutinovic et al.,  
1987a Milutinovic V., Fura D., Helbig W., Linn J. Architecture Compiler Synergism in GaAs Computer Systems//Computer. -1987. -Vol. 20, N 5.- P. 72-93.
- Milutinovic et al.,  
1987b Milutinovic V., Lopez-Benitez N., Hwang K. GaAs-Based Microprocessor Architecture for Real-Time Applications // IEEE Trans. - 1987.- Vol. C-36, N 6.- P. 714-727.
- Minker,  
1971 Minker J. An Overview of Associative or Content-Addressable Memory Systems and KWIC Index to the Literature 1956-1970 // Computing Rev.- 1971, Oct.- P. 453-504.
- Minsky,  
1973 Minsky N. Representation of Binary Trees on Associative Memories // Inform. Proc. Letters.- 1973.- Vol. 2.- P. 1-5.
- Mishra et al.,  
1985 Mishra B., Clarke E. Hierarchical Verification of Asynchronous Circuits Using Temporal Logic // Theor. Comput. Sci. - 1985.- Vol. 38.- P. 269-291.
- Mokhoff,  
1984 Mokhoff N. Parallelism Makes Strong Bid for Next Generation Computers // Computer Des. - 1984, Sep.- P. 104-131.
- Mokhoff,  
1986 Mokhoff N. New RISC Machines Appear as Hybrids With Both RISC and CISC Features // Computer Des. -1986, Apr.- P. 22-25.
- Mokhoff,  
1987 Mokhoff N. Parallelism Breeds a New Class of Supercomputers // Computer Des.- 1987. -Vol. 26, N 6.- P.53-64.



- Moldovan et al., 1985 Moldovan D.I., Tung Y.W. SNAP: A VLSI Architecture for Artificial Intelligence Processing // J. Parallel and Distributed Computing.- 1985.- Vol. 2, N 2.- P. 109-131.
- Moon, 1984 Moon D.A. Garbage Collection in a Large Lisp System // Conf. Rec.1984 ACM Symp. Lisp and Functional Programming.- 1984.- P. 235-246.
- Moon, 1985 Moon D.A. Architecture of the Symbolics 3600//SIGARCH Newsletter. - 1985.- Vol.13, N 3.- P. 76-83.
- Moon, 1986 Moon D. Object-Oriented Programming with Flavors // ACM. Special Issue SIGPLAN Notice. - 1986. - Vol. 21, N 11. - P. 1-8.
- Moon, 1987 Moon D.A. Symbolics Architecture // Computer. - 1987. - Vol. 20, N 1.- P. 43-52.
- Moto-oka et al., 1981 Moto-oka T. et al. Challenge for Knowledge Information Processing Systems. (Preliminary Report on 5th Generation Computer Systems) // Proc. Int. conf. 5th Generation Computer Systems. Oct. 19-20, 1981, Japan Information Processing Development Center.- P. 1-1 - 1-85.
- Mukhopadhyay, 1979 Mukhopadhyay A. Hardware Algorithms for Nonnumeric Computation // IEEE Trans.- 1979.- Vol. C-28, N 6.- P. 384-394.
- muLISP, 1985 muLISP - 8 5. - Reference Manual. - Soft Ware house, Inc., 1985. - 137 p.
- Murakami et al., 1983 Murakami K. et al.. A Relational Data Base Machine: First Step to Knowledge Base Machine // SIGARCH Newsletter. - 1983. - Vol. 11, N 3. - P. 423-425.
- Murakami et al., 1984 Murakami K., Kakuta T., Onai R. Architectures and Hardware Systems: Parallel Inference Machine and Knowledge Base Machine // Proc. Int. Conf. 5th Generation Computer Systems.- 1984.- ICOT, ed.- P. 18-36.
- Murakami et al., 1985 Murakami K., Kakuta T., Onai R., Ito N. Research on Parallel Machine Architecture 5th Generation Computer Systems // IEEE Computer.- 1985, Jun.
- Myamlin et al., 1986 Myamlin A.N., Smirnov V.K., Golovkov S.L. A Specialized Symbol Processor // 5th Generation Computer Architectures / Ed. J.V.Woods.- Amsterdam, New York, Oxford, Tokyo: North-Holland, 1986.- P.301-320.
- Nakazaki et al., 1985 Nakazaki R. et al. Design of a High-speed Prolog Machine (HPM)// ICOT Technical Memorandum: TM-0105.- Tokyo, Jap.- 1985, Apr.
- Nakazaki, 1985 Nakazaki R. et al., Design of a High-Speed Prolog Machine (HPM) // Proc. 18th Annual Workshop in Microprogramming. - 1985. -P. 279-283.
- Nakazaki et al., 1986 Nakazaki R. et al. Design of a High-speed Prolog Machine (HPM) // SIGARCH Newsletter.- 1986.- Vol. 13, N 3.- P. 191-197.
- Nausea et al., 1987 Nausea B.A., Gilbert B.K. A 32 bit 200 MHz GaAs RISC for High-Throughput Signal Processing Environments // IEEE MICRO.- 1987, Dec.

- Neustrasz,  
1985
- Nelson,  
1980
- Nishikama et al,  
1983
- Niwa et al.,  
1984
- Ohr,  
1986a
- Ohr,  
1986b
- Oliver et al.,  
1979
- Onai et al.,  
1985
- Ono et al.,  
1981
- Osato et al.,  
1983
- Osato et al.,  
1984
- Ozkarahan,  
1985
- Patnaik et al.,  
1986
- Patterson et al.,  
1982
- Patterson et al.,  
1983
- Patterson,  
1985
- Patterson,  
1987
- Neustrasz O.M. An Object Oriented System // Office Automation. Concepts and Tools / Ed. by D.C. Tschritzis. - Springer-Verlag, 1985. - P. 167-189.
- Nelson T. Replacing the Printed Word: A Complete Literary System // Information Processing 80, IFIP. - North-Holland, 1980. - P.1013-1023.
- Nishikawa H. et al. The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine Architecture // Logic Programming Workshop-83. - Universidade Lisboa, 1983. - P. 53-73.
- Niwa K. et al. An Experimental Comparison of Knowledge Representation Schemes // AI Magazine. - 1984. - Vol. 5. - P. 29-36.
- Ohr S. Workstation // Electronic Des.- 1986.- Vol. 34, N 1.- P. 69-78.
- Ohr S. 32-bit RISC Chiprips Through 5 MIPS // Electronic Des. - 1986, March. - P. 27-28.
- Oliver E., Berra B. BELACS - A Database Computer // Proc. Int. Conf. Parallel Processing.- Tampa FLA.- 1979.
- Onai R. et al. Architecture of a Reduction-Based Parallel Inference Machine: PIM-R // New Generation Computing.- 1985.-Vol. 3, N. 2.- P. 197-228.
- Ono K., Suzuki M., Goto E. Improvement of Garbage Collection by Aid of Compiler // J. Inform. Process.- 1981.- Vol.4, N 1.- P. 26-34.
- Osato N., Takeuchi I. An Interactive Lisp Programming System TAO/60 // Rev. Electrical Commun. Labs.- 1983.- Vol. 31, N 5.- P. 699-706.
- Osato N., Takeuchi I., Okuno H.G. TAO: A fast Interpreter-Sentered System on Lisp Machine ELIS // CACM.- 1984.- Vol.3, N 8.- P.140-149.
- Ozkarahan E. Evolution and Implementations of the RAP Database Machine // New Generation Computing.- 1985.- Vol. 3.- P.237-271.
- Patnaik L.M., Govindrajan R., Ramadoss N.S. Design and Performance Evaluation of EXMAN: An Extended Manchester Data Flow Computer//IEEE Trans. - 1986.- Vol. C-35, N 3.- P. 229-244.
- Patterson D.A., Sequin C. A VLSI RISC // Computer. - 1982.- Vol. 15, N 9. - P. 8-21.
- Patterson D.A. et al. Architecture of a VLSI Instruction Cache for a RISC // SIGARCH Newsletter. - 1983.- Vol.11, N 3.- P. 108-116.
- Patterson D.A. Reduced Instruction Set Computers // CACM. - 1985. - Vol. 28, N 1.- P. 8-21.
- Patterson D.A. A Progress Report on SPUR // Computer Archit. News. - 1987. - Vol. 15, N 1. - P. 15-21.

- Petri, 1973 Petri C.A. Concepts of Net Theory // Proc. Symp. and Summer Scholl on Mathematical Foundations of Computer Science.-High Tatras, Czechoslovakia. Math. Inst. Slovak Academy Science.- 1973.- P. 137-146.
- Pleszkin et al., 1986 Pleszkin A.R., Thazhuthaveetil M.J. An Architecture for Efficient Lisp List Access // Comp. Archit. News. - 1986. - Vol. 14, N 2. - P. 191-198.
- Pleszkin et al., 1987 Pleszkin A.R., Thazhuthaveetil M.J. The Architecture of LISP Machines // Computer.- 1987, March.- P.35-44.
- Pleszkin et al., 1987 Pleszkin A.R. et al. WISQ: A Restartable Architecture Using Quenes // Comp. Archit. News. - 1987. -Vol. 15, N 2.- P. 290-299.
- Potter, 1981 Potter J.L. Alternative Data Structures for Lists in Associative Devices // Proc. 1981 Int'l Conf. Parallel Processing.- 1983.- P. 486-491.
- Pradhan, 1985 Pradhan D.K. Fault-Tolerant Multi-Processor Link and Bus Network Architectures // IEEE Trans. - 1985. - N 1.
- Przybylski et al., 1984 Przybylski S. et al. Organization and VLSI Implementation of MIPS // J.VLSI and Comput. Systems.- 1984.- Vol. 1, N 3.- P. 170-208.
- Puttkamer, 1983 Puttkamer E.A. Microprogrammed Lisp Machine// Microprocessing and Microprogramming.- 1983. - Vol. 12. - P. 9-14.
- Raghavendra et al., 1985 Raghavendra C.S., Gerla M., Avizienis A. Reliable Loop Topologies for Large Local Computer Networks // IEEE Trans. 1985.- N 1.
- Ram et al., 1986 Ram A., Patel J.H. Parallel Garbage Collection without Synchronization. Overhead // Computer Archit. News.- Vol.14, N 2. - P. 84-90.
- Ramamoorthy et al., 1985 Ramamoorthy C.V. et al. Genesis - An Integrated Environment for Development and Evolution of Software. - COMPSAC, 1985. -183p.
- Ramamoorthy et al., 1987 Ramamoorthy C.V., Shekhar S., Garg V. Software Development Support for AI Programs // Computer. - 1987. - Vol.20, N 1. - P. 30-40.
- Rasset et al., 1986 Rasset T.L., Niederland R.A., Lane J.H., Geideman W.A. A 32-bit RISC Implementation in Enhancement-Mode JFET GaAs//Computer. -1986. - Vol. 19, N 10.- P. 60-68.
- Recoque, 1980 Recoque A. Survey of Main Trends in Computer Hardware Architecture // Proc. IFIP Congress 80. - North Holland. - 1980. - P. 115-125.
- Rem, 1979 Rem V. Mathematical Aspects of VLSI Design // Proc. Caltech Conf. VLSI.- 1979, Jan.
- Rem et al., 1983 Rem M., van de Snepscheut J.L.A., Udding J.T. Trace Theory and the Definition of Hierarchical Components // Proc. Caltech Conf. VLSI.- 1983.- P. 225-239.
- Robert et al., 1987 Robert , Tate. The Lost Generation? // Datamation. - 1987. - Vol. 33, N 13.- P. 44-5 - 44-12.

- Roelof,  
1987a Roelof B. The Trunsputer // A Microprocessor  
Designed For Parallel Processing / MICRO  
CORNUCOPIA. - 1987. - N 38. - P. 6-8.
- Roelof,  
1987b Roelof B. The Trunsputer // The Communication  
Manager / Micro CORNUCOPIA. - 1987.- N 38.-  
P. 10-13.
- Rosenblum et al.,  
1985 Rosenblum L.Ya., Yakovlev A.V. Signal Graphs:  
From Self-Timed to Timed Ones // Proc. Intern.  
Workshop Timed Petri Nets (Torino, Italy, 1985).  
- P. 199-207.
- Ross et al.,  
1975 Ross D.T., Goodenough J.B., Irvine C.A. Software  
1975 Engineering: Process, Principles and Goals  
// Computer. - 1975. - Vol.8, N 5. - P. 17-28.
- Sacerdoti,  
1975 Sacerdoti E. D. A Structure for Plans and  
Behavior. - California: SRI Intern., Menlo Park,  
AI Center, TN 109. - 1975.
- Sacerdoti,  
1976 Sacerdoti E. D. et al. QLISP: A Language for the  
Interactive Development of Complex System // AFIPS  
Conf. Proc. - 1976.- Vol. 45.- P.349-356.
- Samelson et al.,  
1960 Samelson K., Bauer F. Sequential formula  
translation // CACM.- 1960.- Vol.3,N 2.-P.76-83.
- Samples et al.,  
1986 Samples A.D., Hilfinger P., Ungar D. SOAR:  
Smalltalk Without Bytecodes//ACM SIGPLAN  
Notices. - 1986. - Vol 21, N 11. - P. 107-118.
- Schaffert et al.,  
1986 Schaffert C., Cooper T., et al. An Introduction  
to Trellis/Owl // ACM. SIGPLAN Notices.-1986.  
- Vol. 21, N 11. - P. 9-16
- Seitz,  
1980 Seitz C.L. System Timing // Introduction to VLSI  
Systems / Ed. by C.Mead, L.Conway. -  
Addison-Wesley, Reading, MA, 1980.- 400 p.
- Shapiro et al.,  
1983 Shapiro E., Takeuchi A., Object Oriented  
Programming in Concurrent Prolog // New  
Generation Computing.- 1983.- Vol.1.- P.25-48
- Shapiro,  
1986 Shapiro E. Concurrent Prolog: A Progress Report  
// Computer.-1986. - Vol. 19, N 8.- P. 44-58.
- Shimada et al.,  
1986 Shimada T. et al. Evaluation of a Prototype Data  
Flow Processor of the SIGMA-1 for Scientific  
Computation // Computer Archit. News.-  
1986.-Vol.14, N 2.- P.226-234.
- Shobatake et al.,  
1986 Shobatake Y., Aiso H. A Unification Processor  
Based on a Uniformly Structured Cellular  
Hardware // Computer Archit. News. - 1986. -  
Vol.14, N 2.- P. 140-148.
- Short,  
1987 Short B.K. Use of instruction set simulators to  
evaluate the LOW RISC // Computer Archit. News.  
- 1987. - Vol. 15, N 1. - P. 63-67.
- Shuster et al.,  
1979 Shuster S. et al.. RAP.2 - An Associative  
Processor for Data Bases and its Applications //  
IEEE Trans. - 1979. - Vol. C-28.- P. 379-388.
- Siegel,  
1979 Siegel H.I. A Model of SIMD Machines and a  
Comparison of Various Inter-Connection Networks  
// IEEE Trans. - 1979.- N 12.
- Silbey et al.,  
1986 Silbey A., Milutinovic V., Mendoza-Grado V. A  
Sarvey of Advanced Microprocessors and HLL  
Computer Architectures // Computer. - 1986,  
Aug.- P. 72-85.

- Snepscheut, 1983 Snepscheut J.L.A. van de Deriving Circuits from Programs // Proc. Caltech Conf. VLSI.- 1983.- P. 241-256.
- Sohi et al., 1985 Sohi G.S., Davidson E.S., Datel J.H. An Efficient Lisp-Execution Architecture With a New Representation for List Structures // SIGARCH Newsletter.- 1985.- Vol. 13, N 3.- P. 91-98.
- Somani et al., 1984 Somani A.K., Agarwal V.K. An efficient VLSI Dictionary Machine // SIGARCH Newsletter. - 1984. - Vol. 12, N 3.- P. 142-150.
- Sowa, 1987 Sowa M. A Method for Speeding up Serial Processing in Dataflow Computers by Means of a Program Computer // Computer J. - 1987. - Vol. 30, N 4.- P.289-294.
- Stanfill et al., 1986 Stanfill C., Kahle B. Parallel Free-text Search on the Connection Machine System//CACM.- 1986. - N 12. - P. 1229-1239.
- Stanley et al., 1987 Stanley T.J., Wedig R.G. A Performance Analysis of Automatically Managed Top of Stack Buffers // SIGPLAN Notices. - 1987. - Vol. 22, N 10. - P. 272-281.
- Steele et al., 1980 Steele G.L., Sussman G.J. Design of a Lisp-based Microprocessor // CACM.-1980.- Vol. 23, N 11.- P. 628-644.
- Steele et al., 1981 Steele G.L., Sussman G.J., Holloway J., Bell A. Scheme-79 - Lisp on a Chip // Computer.- 1981, Jul.- P. 10-21.
- Steele, 1982 Steele G.L. An Overview of Common Lisp // Proc. Symp. Lisp and Functional Progr.- 1982, Aug.- P. 98-107.
- Steele, 1984 Steele G.L. Common Lisp // Language Digital Press.-1984.- 465 p.
- Steenkiste et al., 1986 Steenkiste P., Hennessy J. LISP on a Reduced-Instruction -Set-Processor // Proc. Conf. LISP and Functional.Progr., ACM, Boston.- 1986, Aug.- P. 192-201.
- Steenkiste et al., 1987 Steenkiste P., Hennessy J., Tags and Type Checking in LISP: Hardware and Software Approaches // SIGPLAN Notices. -1987.-Vol.22, N 10.- P. 50-59.
- Stefic et al., 1983 Stefic M. et al. Knowledge Programming in LOOPS: Report on Experimental Course //AI Magazine. - 1983. - Vol.4, N 3. P. 321-329.
- Sterling et al., 1986 Sterling L. Shapiro E. The Art of Prolog: Advanced Programming Techniques. - MIT Press, 1986. - 427 p.
- Stolfo, 1987 Stolfo S.J. Initial Performance of the DADO2 Prototype // Computer. - 1987. - N 1.
- Stone, 1987 Stone H.S. Parallel Querying of Large Database: A Case Study // Computer. - 1987, Oct.- P.11-21.
- Stoyan, 1978 Stoyan H. LISP-Programmier Handbuch. Eine Sprache für die Nichtnumerische Informationsverarbeitung. - Berlin: Technische Universität, Dresden, Akademie-Verlag, 1978, - 173 p.

- Sumner, 1974 Sumner F.H. MU5 - an Assessment of the Design// Int. Proc. - North-Holland, Amsterdam. - 1974. - P. 133-156.
- Sussman et al., 1971 Sussman G. et al., Microplanner. - Reference Manual. - MA:Cambridge,MIT, AI Lab., Memo 203. - 1971. - 73 p.
- Sussman, 1973 Sussman G. J. A Computation Model of Skill Acquisition. - Mass: MIT, AI Lab., AITR-297. - 1973.
- Sussman et al., 1981 Sussman G.J. et al. Scheme-79 - Lisp on a Chip // Computer.- 1981, July- P. 10-21.
- Taki et al., 1987 Taki K. et al. Performance and Architectural Evaluation of the PSI Machine//SIGPLAN Notices. - 1987. - Vol.22, N 10.- P.128-135.
- Tanaka et al., 1986 Tanaka J., Ueda K., Miyazaki T., Takeuchi A., Matsumoto Y., Furukawa K. Garded Horn Clauses And Experiences With Parallel Logic Programming // Proc. ACM/IEEE Computer Society Fall Joint Computer Conference. - 1986. - P. 948-954.
- Taylor et al., 1986 Taylor G.S. et al. Evaluation of the SPUR Lisp Architecture // Computer Archit. News. - 1986.- Vol. 14, N 2. - P. 444-452.
- Teitelman et al., 1981 Teitelman W. Masinter L. The Interlisp Programming Environment // Computer. - 1981.- Vol.14, N 4. - P. 25-34.
- Thacker et al., 1987 Thacker C.P., Stewart L.C. Firefly: a Multiprocessor Workstation // SIGPLAN Notices.- 1987.- Vol.22, N 10.- P. 164-172.
- Thomas, 1987 Thomas S. LISP Links // Systems International. - 1987. - N 5.- P. 43-48.
- Tick et al., 1984 Tick E., Warren D.H. Towards a Pipelined Prolog Processor // IEEE Computer Society. - 1984, Febr. - P. 29-40.
- Togai, 1986 Togai M., Watanabe H. VLCI Implementation of a Fuzzy Inferent Engine: Toward an Expert System on a Chip // Information Science.- 1986. - Vol. 38, N 2.
- Tomita et al., 1986 Tomita S. et al. Computer with Low-Level Parallelism QA-2 - its Application to 3-D Graphics and Prolog/Lisp Machines//Computer Archit. News. - 1986. - Vol. 14, N 2.- P. 280-289.
- Tong, 1978 Tong R. Synthesis of fuzzy models for industrial processes - some recent results // Int. J. General Systems.- 1978.- 4.
- Tucker, 1986 Tucker M. The AI Single Chip Gets Real // Mini-micro Systems.- 1986.- Vol. 19, N 4.- p.34
- Turchin, 1986 Turchin V.F. The Concept of Supercompiler // ACM Trans. on Programming Languages and Systems. - 1986. - Vol. 8, N 3. - P. 292-325.
- Ungar et al., 1984 Ungar D. et al. Architecture of SOAR: Smalltalk on a RISK // SIGARCH Newsletter. - 1984. - Vol. 12, N 3.- P. 188-197.
- Urmi, 1976 Urmi J. INTERLISP/370. - Reference Manual. - Linkoepping: Linkoepping University, 1976. - 192p.

- van Emden,  
1984
- Veen,  
1986
- Vegdahl,  
1984
- Verity,  
1986
- Wadler,  
1976
- Warren,  
1977
- Warren,  
1980
- Warren,  
1983
- Waters,  
1985
- Weiss,  
1986
- Weiss,  
1987
- Weste,  
1987
- Weston et al.,  
1978
- Whitby-Strevens,  
1985
- White,  
1980
- White,  
1987
- Wielinga et al.,  
1986
- Williams,  
1978
- van Emden M. An Interpreting Algorithm for PROLOG Programs//Implementation ROLOG/Ed. by J.A.Campbell. - Ellis Horwood, 1984. - P. 93-100.
- Veen A.H. Data-flow Machine Architecture//ACM Computing Surveys.-1986.- Vol.18, N 4.- P. 365-396.
- Vegdahl S.R. A Survey of Proposed Architectures for the Execution of Functional Languages//IEEE Trans. - 1984.- Vol. C-33, N 12. - P. 1050-1071.
- Verity J.W. The LISP race heats up // Datamation.-1986.- Vol.32, N 15.- P. 55-58.
- Wadler P.L. Analysis of an Algorithm for Real Time Garbage Collection // CACM.- 1976.- Vol. 19, N 9.- P. 491-500.
- Warren D.H.D. Implementing Prolog - Compiling Predicate Logic Programs // DAI Research Report. - 1977. - Vol. 1,2, N 39,40.
- Warren D.H.D. An Improved Prolog Implementation which Optimises Tail Recursion, Rep. N 156, Department AI, University of Edinburgh, 1980. - 58p.
- Warren D.H.D. An Abstract Prolog Instruction Set. - California: AI Center, 1983. - P. 196.
- Waters R.C. The Programmer's Apprentice: A Session with KBEmacs // TEE Trans. - 1985. - Vol. SE-11, N 11. - P. 233-241.
- Weiss R. Specialized CEPVS Emerge for High-level Languages // Electronic Des. -1986. - N 6.
- Weiss R. RISC Processors: the New Wave in Computer Systems // Computer Des. - 1987. - Vol.26, N 10.- P. 53-73.
- Weste N. Lisp Chips Increase Power With Symbolic Processing // Computer Des.- 1987, Oct.- P.83-87.
- Weston C.D., Stewart G.A. Workstations // Byte. - 1987, Febr - P. 85-97.
- Whitby-Strevens C. The Transputer // SIGARCH Newsletter.- 1985.- Vol. 13, N 3.- P. 292-300.
- White J.L. Address/memory Management for a Gigantic LISP Environment, or GC Considered Harmful // Conf. Rec. LISP Conference (Stanford, Aug. 25-27 1980). - The LISP Company. - 1980.- P. 119-127.
- White R.M. The Role of Symbolic Processing in Supercomputing // Nuclear Simulation/Proc.Int. Symp. and Workshop.- 1987, Oct.-Schliersee, West Germany.- Springer Verlag, 1987.
- Wielinga B., Breuker J. Models of Expertise // Proc. EIJAI-86. - Brighton, 1986. - P. 1123-1125.
- Williams R. A Multiprocessing System for the Direct Execution of Lisp // SIGARCH.- 1978.- Vol. 7, N 2.- P. 35-41.

- Wilson, 1983a  
Wilson, 1983b  
Winograd, 1972  
Winston et al., 1984  
Wise, 1986  
Woo, 1985  
Woods, 1970  
Woods, 1980  
Yamamoto, 1981  
Yao et al., 1987  
Yasuhiko et al., 1986  
Yen et al., 1982  
Yokota et al., 1986  
Yonezawa et al., 1986  
Yuhara et al., 1986  
Zagorulko, 1984  
Zamenek, 1971
- Wilson P. Occam Architecture Eases System Design Pt. I // Computer Des. - 1983, Nov. - P. 107-115.  
Wilson P. Language-based Architecture Eases System Design - Pt. II // Computer Des. - 1983, Dec. - P. 109-120.  
Winograd T. Understanding Natural Language. Edinburg Univ. Press., 1972. - 127 p.  
Winston P.H., Horn B.K.P. Lisp. - Addison-Wesley Publishing Company, 1984. - Second edition. - 153 p.  
Wise M. J. , Prolog Multiprocessors. - New York: Prentice Hall, 1986. - 168 p.  
Woo N.S. A Hardware Unification Unit: Design and Analysis // SIGARCH Newsletter. - 1985. - Vol. 13, N 3. - P. 198-205.  
Woods W. Transition Network Grammar for Natural Language Analysis // CACM - 1970. - Vol. 13. - P. 86-93.  
Woods W. Cascaded ATN grammars // American J. Comp. Linguistics. - 1980. - Vol. 6, N 1. - P. 123-141.  
Yamamoto M. A Survey of High-level Language Machines in Japan // Computer. - 1981. - Vol. 14, N 7. - P. 68-78.  
Yao S.B., Hevner A.R., Young-Myers H. Analysis of Database System Architectures Using Benchmarks // IEEE Trans. - 1987. - Vol. SE-13, N 6. - P. 709-725.  
Yasuhiko Y., Tokoro M. The Design and Implementation of Concurrent Smalltalk // ACM Special Issue of SIGPLAN Notice. - 1986. - Vol. 21, N 11. - P. 331-340.  
Yen D.W.L., Kulkarni A.V. Systolic Processing and an Implementation for Signal and Image Processing // IEEE Trans. - 1982. - Vol. C-31, N 10. - P. 1000-1009.  
Yokota H., Itoh H.A. Model and Architecture for a Relational Knowledge Base // Computer Arch. News. - 1986. - Vol. 14, N 2. - P. 2-19.  
Yonezawa A., Briot J., Shibayama E. Object-Oriented Concurrent Programming in ABCL/1 // ACM Special Issue of SIGPLAN Notice - 1986. - Vol 21, N 11. - P. 224-231.  
Yuhara M. et al. Evaluation of the FACOM ALPHA Lisp Machine // Comput. Arch. News. - 1986. - Vol. 14, N 2. - P. 184-190.  
Zagorulko Yu.A. A Program System for Efficient Operation With Embedded Semantic Network // Computers and Artificial Intelligence. - 1984. - Vol. 3, N 6. - P. 483-494.  
Zamenek H. Summation and Future Directions of Associative Information Techniques // Associative Information Techniques / Ed. by E.L.Jacks. - N.Y.: American Elsevier. - 1971. - P. 205-215.



## ОГЛАВЛЕНИЕ

Предисловие (В. Н. Захаров, В. Ф. Хорошевский)	5
<b>Глава 1. Базовые средства программирования для интеллектуальных систем</b>	<b>7</b>
1.1. Средства поддержки разработки интеллектуальных систем (А. Г. Красовский, В. Ф. Хорошевский)	7
1.2. Языковые средства программирования (В. Ф. Хорошевский)	17
1.3. Язык Лисп и его модификации (О. В. Ковригин, К. Г. Перфильев)	21
1.4. Язык Пролог и методы его реализации (Н. И. Ильинский, И. Б. Козинцев)	33
1.5. Метаалгоритмический язык Рефал и тенденции его развития (С. А. Романенко)	48
1.6. Языки программирования интеллектуальных решателей (В. Н. Лихолит, В. Н. Пильщиков)	56
1.7. Система Smalltalk-80 и объектно-ориентированное программирование (Д. И. Безруков, А. О. Голосов)	67
<b>Глава 2. Языки и системы представления знаний</b>	<b>72</b>
2.1. Программные средства представления знаний: состояние исследований и проблемы (В. Ф. Хорошевский)	72
2.2. Язык представления знаний оболочки СПЭИС (О. В. Ковригин)	82
2.3. Язык представления знаний Пилот (В. Ф. Хорошевский, С. Ю. Щенников)	87
2.4. Язык представления лингвистических знаний ATNL-2.0 (В. Ф. Хорошевский)	95
2.5. Средства для работы со знаниями в технологическом комплексе ТХК-БЗ (Е. Ю. Кандрашина)	103
<b>Глава 3. Инструментальные средства для разработки интеллектуальных систем</b>	<b>109</b>
3.1. Технологические комплексы для разработки баз знаний (Е. Ю. Кандрашина)	109
3.2. ПРИЗ — семейство инструментальных средств, ориентированных на знания (А. Калья, М. Коов, М. Кыпп, М. Мацкин, Я. Пеньям, Ф. Перкманн, Э. Тыгуу, Ф. Хаав, А. Шмундак)	120
3.3. Инструментальный технологический комплекс для систем семиотического моделирования (А. А. Абрахманов, В. С. Лозовский, С. В. Лозовский)	131
3.4. Средства поддержки проектирования прикладных экспертных систем в оболочке СПЭИС (О. В. Ковригин)	151
3.5. ПИЭС — программный инструментарий для экспертных систем (А. Ю. Алешин, В. Ф. Хорошевский, В. Ю. Шерстнев, С. Ю. Щенников)	155

<b>Глава 4. Принципы разработки аппаратных средств поддержки интеллектуальных систем</b>	<b>168</b>
4.1. Аппаратная реализация интеллектуальных систем (В. Н. Захаров, Л. К. Эйсымонт)	168
4.2. Элементная база интеллектуальных систем (В. Н. Захаров, Л. К. Эйсымонт)	181
4.3. Аperiodическая схемотехника (В. И. Варшавский, В. Б. Мараховский, Л. Я. Розенблюм, А. В. Яковлев)	199
<b>Глава 5. Специализированные процессоры для интеллектуальных систем</b>	<b>213</b>
5.1. Машины баз данных (М. М. Гилула, Л. А. Калинин, С. К. Ландо, В. М. Рывкин)	213
5.2. Ассоциативные параллельные процессоры (Е. К. Гордиенко)	235
5.3. Однородные структуры (В. Н. Захаров, В. Ю. Кириллов)	249
5.4. Специализированные вычислительные структуры (А. П. Горяшко)	258
5.5. Средства обработки нечеткой информации (Л. С. Берштейн, С. Я. Коровин, А. Н. Мелихов)	284
<b>Глава 6. Специализированные процессоры для языков высокого уровня</b>	<b>293</b>
6.1. Лисп-процессоры (А. Н. Мямлин, А. Г. Рубин, В. К. Смирнов)	293
6.2. Пролог-машины и спецпроцессоры вывода (В. Н. Вагин, В. Н. Захаров)	312
6.3. Реал-процессор (С. Л. Головков, В. К. Смирнов)	321
<b>Список литературы</b>	<b>329</b>

Справочное издание

**ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ: В 3-х кн.**

Книга 3

**ПРОГРАММНЫЕ И АППАРАТНЫЕ СРЕДСТВА**

ПОД РЕД. В. Н. ЗАХАРОВА, В. Ф. ХОРОШЕВСКОГО

Справочник

Заведующая редакцией Г. И. Козырева  
Редакторы Т. М. Толмачева, Т. М. Любимова  
Переплет художника Н. А. Пашуро  
Художественный редактор Н. С. Шейн  
Технический редактор Л. А. Горшкова  
Корректор Т. В. Дземидович

**ИБ № 2231**

Сдано в набор 29.01.90 Подписано в печать 29.05.90 Т-06965 Формат 60×88/16  
Бумага тип. № 2 Гарнитура литературная Печать офсетная Усл. печ. л. 22,54  
Усл. кр.-отт. 22,54 Уч. изд. л. 33,23 Тираж 25000 экз. Изд. № 22879 Зак. № 6939 Цена 2 р.

Издательство «Радио и связь». 101000 Москва, Почтамт, а/я 693

Ордена Октябрьской Революции и ордена Трудового Красного Знамени МПО «Первая  
Образцовая типография» Государственного комитета СССР по печати: 113054, Москва,  
Валовая, 28.