

**БИБЛИОТЕЧКА  
ПРОГРАММИСТА**

**А.Н. АНДРИАНОВ  
С.П. БЫЧКОВ  
А.И. ХОРОШИЛОВ**

# **Программирование на языке симула-67**



# БИБЛИОТЕЧКА ПРОГРАММИСТА

---

А. Н. АНДРИАНОВ,  
С. П. БЫЧКОВ,  
А. И. ХОРОШИЛОВ

## ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИМУЛА-67

Под редакцией А. Н. МЯМЛИНА



МОСКВА «НАУКА»  
ГЛАВНАЯ РЕДАКЦИЯ  
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ  
1985

## ОГЛАВЛЕНИЕ

Предисловие редактора . . . . .	5
Предисловие . . . . .	7
<b>Глава 1. Понятие объекта в языке симула-67 . . .</b>	<b>9</b>
1.1. Объекты и классы объектов . . . . .	9
1.2. Создание объектов и их именование . . . . .	13
1.3. Иерархическое описание классов объектов . . . . .	17
1.4. Средства доступа к атрибутам объектов . . . . .	26
1.5. Организация упорядоченных множеств объектов . . . . .	32
1.6. Динамическое взаимодействие объектов. Сопрограммы . . . . .	42
<b>Глава 2. Средства имитационного моделирования . .</b>	<b>50</b>
2.1. Процессы. Структура класса simulation . . . . .	50
2.2. Операторы управления процессами . . . . .	58
2.3. Структура программы моделирования . . . . .	68
2.4. Средства стохастического моделирования . . . . .	78
2.5. Процедуры для сбора статистики . . . . .	83
2.6. Примеры программ моделирования на языке симула-67 . . . . .	87
<b>Глава 3. Обработка текстов . . . . .</b>	<b>130</b>
3.1. Литеры . . . . .	130
3.2. Тексты . . . . .	133
3.3. Ввод-вывод в языке симула-67 . . . . .	152
<b>Глава 4. Работа с симула-программами на БЭСМ-6 и ЕС ЭВМ . . . . .</b>	<b>168</b>
4.1. Трансляторы с языка симула-67 для БЭСМ-6 и ЕС ЭВМ . . . . .	168
4.2. Входной язык трансляторов . . . . .	182
4.3. Раздельная компиляция симула-программ . . . . .	188
4.4. Средства связи с программами на других языках . . . . .	198
4.5. Отладка симула-программ в пакетном режиме . . . . .	202
4.6. Диалоговая отладка симула-программ . . . . .	216
<b>Глава 5. Применение языка симула-67 для разработки пакетов прикладных программ . . . . .</b>	<b>226</b>
5.1. Использование иерархий деклараций классов при разработке ППП . . . . .	226

5.2. Пакет для моделирования непрерывно-дискретных систем . . . . .	232
5.3. Библиотека процедур и классов для построения графиков . . . . .	236
5.4. Пакет программ для сбора статистики . . . . .	240
Приложение 1. Отличия языка симула-67 от алгола-60 . . . . .	244
Приложение 2. Правила передачи параметров . . . . .	244
Приложение 3. Сообщения о динамических ошибках . . . . .	247
Приложение 4. Сообщения компиляторов . . . . .	252
Приложение 5. Кодировка основных символов и операций для трансляторов на БЭСМ-6 и ЕС ЭВМ . . . . .	257
Приложение 6. Таблицы рангов литер для БЭСМ-6 и ЕС ЭВМ . . . . .	258
Приложение 7. Сообщения диалогового отладчика . . . . .	259
Приложение 8. Пример диалога . . . . .	263
Приложение 9. Примеры симула-программ . . . . .	270
9.1. Программа расстановки ферзей . . . . .	270
9.2. Программа класса SIMTAPL . . . . .	275
9.3. Программа класса DISCONT . . . . .	279
9.4. Программа использования классов SIMTAPL и DISCONT . . . . .	281
Список литературы . . . . .	286

## ПРЕДИСЛОВИЕ РЕДАКТОРА

Наблюдающийся в последние десятилетия быстрый рост производительности ЭВМ является материальной основой для успешного использования метода имитационного моделирования при исследовании и проектировании сложных систем в самых различных предметных областях. Модельный эксперимент позволяет изучать объекты, над которыми прямой эксперимент затруднен, экономически невыгоден либо вообще невозможен в силу тех или иных причин (моделирование летательных аппаратов, сложных промышленных комплексов, экономических систем, процессов в микро- и макромире, боевых действий и т. д.). Однако разработка моделей сложных многопараметрических систем является трудоемкой задачей и требует высокой квалификации не только в предметной области, а также и в области разработки программного обеспечения.

Одним из путей снижения трудоемкости разработки имитационных моделей является создание узкоспециализированных языков моделирования и пакетов прикладных программ, ориентированных на применение в конкретной предметной области или решение некоторого класса задач. Эти языки и пакеты должны в удобной для пользователя форме отображать понятия и методы, выработанные в данной предметной области, а также содержать модели, описывающие типовые свойства характерных для нее объектов. Для специализированных языков моделирования и пакетов типовых моделей весьма желательна возможность включения новых понятий и конкретизации описания типовых свойств, позволяющая отслеживать процесс постепенной, поэтапной детализации представлений о моделируемых системах в ходе их исследования или проектирования. Это требование связано с тем, что предварительный учет всех параметров, определяющих поведение исследуемого объекта, часто бывает невозможен, так как модель является одновременно инструментом и объектом экспериментального исследования. Поэтому чрезвычайно важно иметь возможность сравнительно просто совершенствовать модель в процессе изучения объекта.

Языком высокого уровня, получившим широкое применение для решения задач имитационного моделирования, является язык симула-67, разработанный в Норвежском вычислительном центре. В нем впервые получила практическое воплощение концепция языка — ядра. Эта особенность языка позволяет сравнительно просто вводить расширения языка, ориентированные на конкретные области применения, т. е. вводить объекты и

средства их описания, привычные для специалиста, работающего в данной области.

Язык симула-67 реализован на многих зарубежных ЭВМ и отечественных ЭВМ БЭСМ-6 и ЕС ЭВМ. Близки к завершению работы по созданию транслятора с этого языка на МВК Эльбрус.

Компиляторы с языка симула-67 для ЕС ЭВМ и БЭСМ-6, разработанные авторами книги, успешно используются в ряде организаций для моделирования сложных систем, включая вычислительные системы, сети ЭВМ, робототехнические комплексы, системы передачи информации, для разработки расширяемых специализированных пакетов прикладных программ и т. д.

Книга содержит неформальное описание основных понятий языка симула-67 и некоторые приемы программирования. Приведенные правила эксплуатации компиляторов с языка симула-67 для ЭВМ БЭСМ-6 и ЕС ЭВМ значительно облегчают практическое применение языка.

Книга рассчитана на читателя, знакомого с основами программирования и знающего язык алгол-60. Она может служить учебником по языку симула-67 и адресуется широкому кругу пользователей ЭВМ, занимающихся проектированием и моделированием сложных систем, а также студентам и аспирантам вузов.

*А. Н. Мямлин*

## ПРЕДИСЛОВИЕ

Симула-67 [10] — это универсальный язык программирования, в котором получили дальнейшее развитие основные понятия алгоритмического языка алгол-60, входящего в симула-67 как подмножество, и языка моделирования симула-1 [33]. В дополнение к средствам алгола-60 язык симула-67 содержит удобный аппарат описания новых понятий, средства обработки текстовой информации, стандартные средства ввода-вывода, средства для организации квазипараллельного исполнения компонентов программы.

Важным достоинством языка симула-67 является возможность построения на его основе и его средствами специализированных языков программирования и пакетов прикладных программ, содержащих основные понятия, выработанные в некоторой (возможно, довольно узкой) предметной области. Такое специализированное программное обеспечение значительно облегчает процесс общения с ЭВМ специалистов, работающих в этой предметной области, и служит средством фиксации опыта программирования возникающих в ней задач. В языке симула-67 имеются специальные средства, ориентированные на удобное отображение процесса постепенной детализации и конкретизации понятий и действий, происходящего в ходе исследований и разработок, проводимых в данной предметной области.

Язык симула-67 реализован на отечественных ЭВМ БЭСМ-6 и ЕС ЭВМ [1, 7], а также на ряде типов ЭВМ, выпускаемых ведущими зарубежными фирмами по производству вычислительной техники [41]. Наиболее широко симула-67 используется для решения задач имитационного моделирования сложных систем. Средства моделирования, ориентированные на представление системы в виде совокупности взаимодействующих процессов, определены в системном (т. е. реализуемом вместе с транслятором) классе *simulation*. Описание этих средств на языке симула-67, имеющееся в руководствах по языку [10, 35, 36], служит хорошим примером построения специализированного программного обеспечения для конкретной предметной области. С помощью языка симула-67 успешно разрабатываются модели

вычислительных систем [3, 4], сетей ЭВМ [12], бортовых систем управления, использующих БЦВМ [6], робототехнических систем [21]. Имеется опыт применения языка симула-67 для разработки специализированных средств моделирования для описания систем с потоками заявок [23, 24], непрерывно-дискретных систем [2, 37], систем искусственного интеллекта [15].

В предлагаемой работе на неформальном уровне рассматриваются средства языка симула-67, выходящие за рамки алгола-60. Основное внимание уделяется средствам структурированного описания сложных объектов и понятий, средствам моделирования и методике их применения.

В главе 1 вводится центральное понятие языка симула-67 — понятие объекта, рассматриваются средства иерархического описания классов объектов, аппарат именования объектов и механизмы доступа к их атрибутам, базовые средства организации квазипараллельной работы объектов.

Глава 2 посвящена средствам моделирования, определенным в системном классе *simulation*, примерам их применения при разработке имитационных моделей.

В главе 3 рассматриваются средства языка симула-67 для обработки текстовой информации и организации ввода-вывода.

Глава 4 содержит описание особенностей входных языков компиляторов с языка симула-67 для ЭВМ БЭСМ-6 и ЕС ЭВМ, правил их эксплуатации и сервисных возможностей, предоставляемых системами программирования на базе языка симула-67.

В главе 5 рассматривается методика применения языка симула-67 для разработки пакетов прикладных программ и специализированных языков программирования, описывается ряд конкретных пакетов, разработанных с использованием языка симула-67.

Авторы считают приятным долгом выразить благодарность своим научным руководителям доц. И. Б. Задыхайло, проф. Л. Т. Кузину, проф. А. Н. Мямлину за постоянное внимание и поддержку в работе по реализации языка симула-67. Искреннюю благодарность авторы выражают И. М. Седовой, Г. В. Попковой, А. Л. Пытелю, А. Е. Шестакову, участвовавшим в разработке трансляторов с языка симула-67, А. А. Веденову, С. А. Романенко, А. Е. Фирсову за помощь в использовании рефал-систем на БЭСМ-6 и ЕС ЭВМ, а также всем пользователям, чьи замечания способствовали совершенствованию трансляторов с языка симула-67 на ЕС ЭВМ и БЭСМ-6.

*А. Н. Андрианов, С. П. Бычков, А. И. Хорошилов*



## ГЛАВА 1

### ПОНЯТИЕ ОБЪЕКТА В ЯЗЫКЕ СИМУЛА-67

Глава посвящена центральному понятию языка симула-67 — понятию объекта. В § 1.1 дается определение объекта и класса объектов, на примере рассматривается принятая в языке форма задания классов объектов; в § 1.2 рассматриваются языковые средства для генерации и именования объектов; в § 1.3 разбирается аппарат иерархического, структуризованного описания классов объектов; в § 1.4 подробно освещается механизм доступа к атрибутам объектов. В качестве примеров применения введенных средств языка симула-67 в § 1.5 рассматриваются вопросы организации упорядоченных множеств объектов и отображения различных типов структурных взаимосвязей объектов. § 1.6 посвящен базовым средствам организации квазипараллельной работы объектов.

#### 1.1. Объекты и классы объектов

Термином *объект* в языке симула-67 обозначаются программные компоненты, обладающие собственными локальными данными (атрибутами) и способные выполнять определенные действия. В роли атрибутов могут выступать переменные, массивы, ссылки на другие объекты, процедуры.

Действия, выполняемые объектом, задаются с помощью последовательности операторов, допустимых в языке симула-67. Эту последовательность часто называют *правилами действий* объекта, или *сценарием* его поведения (функционирования).

Объекты в симула-программах служат удобным средством для отображения существующих или проектируемых объектов реального мира, а также абстрактных понятий. Например, автомобиль, имеющий вес 5,2 т, длину 4 м, прошедший путь в 3260 км и движущийся в соответствии с правилами уличного движения, может быть представлен объектом, который схемати-

чески изображен на рис. 1. Правила действий автомобиля для наглядности представлены неформально, в виде текста на естественном языке.

Каждый объект обладает системным атрибутом, указывающим на исполняемый оператор его правил действий. Этот атрибут называют *локальным управлением* (ЛУ). Во время работы симула-программы могут одновременно существовать несколько

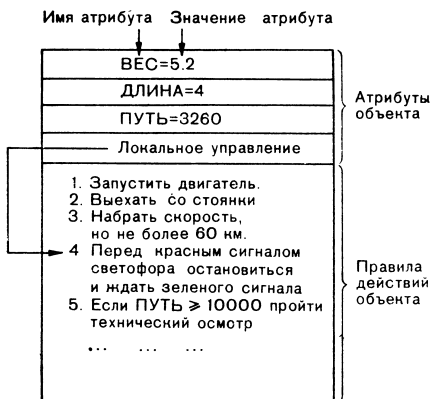


Рис. 1. Структура объекта в языке симула-67.

объектов (их количество ограничивается лишь объемом памяти ЭВМ), находящихся на разных стадиях исполнения своих сценариев функционирования. Среди них в каждый момент времени лишь один объект действительно выполняет операторы правил действий. Этот объект называют *активным*, или *текущим*. Локальное управление текущего объекта совпадает с центральным управлением симула-программы и движется по операторам его правил действий. Локальное управление любого другого объекта при этом остается неподвижным и указывает на оператор, с которого начнется работа этого объекта, когда он станет текущим. Таким образом, в ходе работы симула-программы операторы правил действий объекта могут исполняться за несколько *активных фаз* (так называются моменты времени, когда объект активен), в промежутках между которыми работают другие объекты. Такой способ взаимодействия объектов получил название *квазипараллельного исполнения* [10], а сами объекты в этом случае часто называют *сопрограммами*. Передача управления от одного объекта к другому осуществляется с помощью специальных операторов, которые подробно рассматриваются в § 1.6. Заметим, что саму симула-программу можно рассматривать как объект, получающий управление от операционной системы ЭВМ.

Объекты, присутствующие в симула-программе, принадлежат к одному или нескольким множествам родственных объектов, называемых *классами*. Объекты одного класса имеют общую структуру атрибутов и одинаковые правила действий, но могут различаться конкретными значениями своих атрибутов. Поскольку в программе может быть много объектов одного класса, то удобно один раз описать присущие им структуру атрибутов и правила действий, а затем создавать нужное количество отдельных объектов, указывая конкретные значения их атрибутов. Создание объектов рассматривается в § 1.2.

Классы объектов определяются в программах на языке симула-67 с помощью специальных описаний, называемых *декларациями* классов. Синтаксически декларация класса очень похожа на описание процедуры. Рассмотрим строение декларации класса на примере описания автомобилей, каждый из которых характеризуется весом, длиной, количеством пройденных километров и действует в соответствии с правилами уличного движения (см. рис. 1).

Пример 1.1.

Заголовок декларации класса	<div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> <div style="font-size: 3em; line-height: 1;">{</div> <div style="font-size: 2em; line-height: 1;">{</div> </div> <div> <div style="text-align: center;">имя класса</div> <div>class АВТОМОБИЛЬ (ВЕС, ДЛИНА);</div> <div style="text-align: right;">формальные параметры декларации класса</div> <div>real ВЕС, ДЛИНА; ← спецификация параметров</div> </div> </div>
Тело декларации класса	<div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> <div style="font-size: 3em; line-height: 1;">{</div> <div style="font-size: 2em; line-height: 1;">{</div> </div> <div> <div>begin real ПУТЬ;</div> <div>procedure ТЕХОСМОТР; ... ;</div> <div>procedure СТОП (V); real V; ... ;</div> <div style="text-align: center; margin: 10px 0;">. . . . .</div> <div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> <div style="font-size: 3em; line-height: 1;">{</div> <div style="font-size: 2em; line-height: 1;">{</div> </div> <div> <div>if ПУТЬ ≥ 10000 then</div> <div>ТЕХОСМОТР;</div> <div style="text-align: center; margin: 10px 0;">. . . . .</div> <div>if КРАСНЫЙ then СТОП</div> <div>(60);</div> </div> </div> </div> </div>
	<div style="display: flex; align-items: center;"> <div style="margin-right: 10px;"> <div style="font-size: 3em; line-height: 1;">{</div> <div style="font-size: 2em; line-height: 1;">{</div> </div> <div>end декларация класса АВТОМОБИЛЬ;</div> </div>

Декларация класса состоит из заголовка и тела. В заголовке декларации класса указывается имя класса, перечисляются формальные параметры декларации, т. е. атрибуты, начальные значения которых определяются при генерации объектов данного класса, задаются спецификации этих атрибутов. В роли формальных параметров деклараций классов могут использоваться переменные и массивы, т. е. допускаются спецификации <тип>, **array** и <тип> **array**. Допустимыми фактическими параметрами для формальных переменных являются

выражения, а для формальных массивов — идентификаторы массивов.

Передача фактических параметров для атрибутов со спецификациями **real**, **integer**, **boolean**, **character** (эти типы в языке симула-67 называют обыкновенными) выполняется по значению. Атрибуты ссылочных типов (ссылки на объекты и тексты) и атрибуты-массивы вызываются, как правило, по ссылке, т. е. в тело декларации передается копия ссылки на соответствующий фактический объект (текст, массив).

**З а м е ч а н и е.** Реализации языка симула-67 на БЭСМ-6 [1] и ЕС ЭВМ [7] допускают вызов по наименованию параметров деклараций классов со спецификацией **<тип>**, а также задание параметров со спецификациями **procedure**, **<тип> procedure** и **label**.

Телом декларации класса может быть любой оператор языка симула-67. Чаще всего в роли тела декларации используется блок. Все переменные, массивы, процедуры, классы, определенные в блоке тела декларации, считаются атрибутами объектов определяемого ею класса. От атрибутов-параметров они отличаются лишь тем, что при создании объекта они получают стандартные начальные значения: переменным и элементам массивов типа **real** или **integer** присваивается значение 0, типа **boolean** — **false**, типа **text** — **notext**, типа «ссылка на объект» (**ref**) — **none**.

Важным свойством всех атрибутов объекта является их доступность не только из правил действий этого объекта, но и из других объектов (извне), что позволяет легко организовывать информационное взаимодействие объектов (средства доступа к атрибутам рассматриваются в § 1.4). Отметим, что переменные (массивы, процедуры и т. д.), описанные в блоках, вложенных в тело декларации, извне недоступны.

Рассмотрим еще один, более конкретный, пример использования деклараций классов.

Пусть требуется описать поведение шахматной пешки, которую ставят на произвольную клетку доски. Если эта клетка занята, то пешка пытается встать на клетку, доступную из начальной за один ход. Если и она оказывается занятой, то пешка на доску не ставится. Будем считать, что клетка шахматной доски идентифицируется парой чисел, определяющих номера соответствующей вертикали и горизонтали, и что белые пешки ходят в направлении увеличения (черные — уменьшения) номера горизонтали.

Опишем шахматные пешки, действующие по указанным выше правилам, с помощью декларации класса ПЕШКА, которая задает для них следующие атрибуты: начальная вертикаль (НВ),

начальная горизонталь (НГ), определяющие исходную клетку, текущая вертикаль (ТВ), текущая горизонталь (ТГ), окончательное значение которых определяет занятую пешкой клетку (если поставить пешку нельзя, то атрибутам ТВ и ТГ присваиваются нулевые значения). Булевский атрибут БЕЛАЯ задает цвет пешки.

Пример 1.2.

```
class ПЕШКА (НВ, НГ, БЕЛАЯ); integer НВ, НГ; boolean
БЕЛАЯ;
begin integer ТВ, ТГ;
  ТВ:=НВ; ТГ:=НГ;
  if СВОБОДНА (ТВ, ТГ) then goto ОСТАНОВКА;
  comment исходная клетка занята, делаем ход;
  ТГ:= ТГ + (if БЕЛАЯ then 1 else - 1);
  if (ТГ>8 or ТГ<1) or not СВОБОДНА (ТВ, ТГ)
  then goto ВСТАТЬ НЕКУДА;
  ОСТАНОВКА: ЗАНЯТЬ (ТВ, ТГ); goto КОНЕЦ;
  ВСТАТЬ НЕКУДА: ТВ:= ТГ:= 0; КОНЕЦ;
end класса ПЕШКА;
```

В приведенной декларации класса использованы обращения к булевской процедуре-функции СВОБОДНА, дающей значение **true**, если клетка не занята, и **false** — в противном случае, и к процедуре ЗАНЯТЬ, которая делает занятой клетку, заданную в ее фактических параметрах. Их описания мы не приводим, поскольку в данном случае не важен конкретный способ выполнения задаваемых ими действий.

В том случае, если телом декларации класса является пустой оператор либо блок тела содержит только описания, соответствующие объекты не будут выполнять никаких действий и могут использоваться как пассивные структуры данных, хранящие значения своих атрибутов. Такие объекты аналогичны структурам в языке PL/1 [28] или записям в паскале [9].

## 1.2. Создание объектов и их именование

Декларации классов в программах на языке симула-67 определяют классы объектов, задавая присущие им характерные признаки (структуру атрибутов и правила функционирования), общие для всех объектов данного класса.

Создание конкретного объекта, принадлежащего к определяемому декларацией классу и имеющего определенные значения атрибутов, выполняется с помощью конструкции вида **new** <имя класса> (<совокупность фактических параметров>) называемой *генератором объектов*.

В совокупности фактических параметров задаются начальные значения для тех атрибутов объекта, которые определены в списке формальных параметров декларации класса. Остальные атрибуты принимают при генерации объекта стандартные начальные значения (см. 1.1). Количество фактических и формальных параметров должно быть одинаковым, типы фактических параметров должны, как правило, совпадать с типами соответствующих им формальных параметров (более подробно механизм передачи параметров рассматривается в Приложении 2). Если в декларации класса не заданы формальные параметры, то генерация соответствующих объектов обеспечивается конструкцией вида `new <имя класса>`.

**Пример 1.3.** Генератор объекта

`new ПЕШКА (3, 2, true)`

отображает постановку белой пешки на клетку (3, 2). Декларация класса ПЕШКА приведена в примере 1.2.

Результатом вычисления генератора объектов является ссылка на новый объект указанного в нем класса с конкретными значениями атрибутов. Эта ссылка может быть присвоена переменной (простой или с индексами) типа «ссылка на объект», которая в дальнейшем служит для обращения к сгенерированному объекту и его атрибутам.

Описания переменных, массивов, процедур-функций, а также спецификации параметров, значениями которых могут быть ссылки на объекты, начинаются с указателя типа вида `ref (<имя класса>)`. Конструкция `<имя класса>` в данном случае называется квалификацией определяемых переменных (массивов, процедур, параметров). Роль квалификации состоит в ограничении области возможных значений ссылочных переменных: переменная может ссылаться лишь на объекты класса, указанного в ее квалификации. Это относится, конечно, и к массивам ссылочных переменных, и к процедурам-функциям, выдающим значение типа «ссылка на объект».

**З а м е ч а н и е.** Область возможных значений для ссылочных переменных будет уточнена в § 1.3 после рассмотрения аппарата иерархий деклараций классов. Квалификация ссылок играет важную роль и в средствах доступа к атрибутам объектов (см. 1.4).

Явное задание области возможных значений ссылочных переменных с помощью квалификации позволяет контролировать правильность их использования для именования объектов во время трансляции симула-программы.

**Пример 1.4. а) Описания**

`ref (АВТОМОБИЛЬ) АВТ1, АВТ2;`

`ref (АВТОМОБИЛЬ) array КОЛОННА [1 : N];`

задают переменные АВТ1 и АВТ2 и массив КОЛОННА из N элементов, значениями которых могут быть ссылки на объекты класса АВТОМОБИЛЬ.

б) С помощью следующей процедуры-функции

```
ref (АВТОМОБИЛЬ) procedure САМЫЙ ТЯЖЕЛЫЙ (А, М);  
  ref (АВТОМОБИЛЬ) array А; integer М;  
  begin поиск в массиве А автомобиля с максимальным весом  
  end;
```

можно вычислить ссылку на самый тяжелый автомобиль из тех, на которые ссылаются элементы массива А длины М.

Присваивание ссылочной переменной ссылки на конкретный объект производится с помощью оператора присваивания ссылок, в левой части которого пишется переменная (простая или с индексами), а в правой — генератор объекта (в общем случае объектное выражение). Левая и правая части разделяются знаком : — (двоеточие, минус), который читается как «обозначает».

Например, если мы хотим обозначить через П1 объект, представляющий крайнюю левую белую пешку (см. 1.1, пример 1.2), то для этого нужно задать описание

```
ref (ПЕШКА) П1;
```

и выполнить оператор присваивания ссылок

```
П1 : — new ПЕШКА (1, 2, true)
```

Логическая структура данных, возникающая в памяти ЭВМ после исполнения этого оператора, изображена на рис. 2\*). Отметим, что при создании каждого объекта динамически отводится участок памяти для размещения его атрибутов. Правилам действий всех объектов одного класса соответствует один и тот же участок памяти, в котором располагается изготовленная компилятором реентерабельная машинная программа, соответствующая декларации класса. Однако из соображений наглядности мы и в дальнейшем будем изображать объекты так, как это показано на рис. 1, т. е. имеющими собственную копию правил действий.

Пример 1.5. Начальная расстановка всех черных пешек может быть описана с помощью следующего фрагмента программы:

```
ref (ПЕШКА) array ЧЕРНАЯ ПЕШКА [1 : 8];  
integer I;
```

---

\*) На всех рисунках подчеркнутые слова соответствуют основным символам языка симула-67.

for I:=1 step 1 until 8 do

ЧЕРНАЯ ПЕШКА [I]:—new ПЕШКА (I, 7, false);

Одному объекту в языке симула-67 можно дать несколько имен, присвоив ссылку на этот объект несколькими переменными.

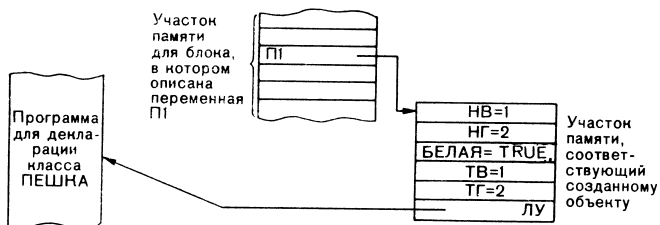


Рис. 2. Логическая структура данных, возникающая при генерации объекта.

Например, четвертой черной пешке (пример 1.5) можно дать еще одно имя, скажем, ЧП4:

ref (ПЕШКА) ЧП4;

ЧП4:—ЧЕРНАЯ ПЕШКА [4];

Употребление переменных ЧП4 и ЧЕРНАЯ ПЕШКА [4] будет теперь означать обращение к одному и тому же объекту.

Созданный объект существует в симула-программе до тех пор, пока на него имеется хотя бы одна ссылка. Уничтожить ссылку на объект можно путем присваивания ссылочной переменной специального значения none — «никто» или «нет объекта». Кроме того, ссылка на объект, хранящаяся в ссылочной переменной, уничтожается при присваивании ей нового значения, а также при выходе из блока, в котором описана эта переменная. С исчезновением последней ссылки на объект занимаемая им память автоматически объявляется свободной и может быть использована для других нужд. Программные комплексы, реализующие язык симула-67, обеспечивают автоматический учет и распределение освобождающихся участков памяти.

Между ссылками на объекты определены два отношения: идентичность ( $=$ ) и неидентичность ( $\neq$ ). Отношение  $X = Y$  истинно в том и только том случае, если ссылочные переменные (в общем случае — объектные выражения)  $X$  и  $Y$  указывают на один и тот же объект или одновременно равны none. Отношение  $X \neq Y$  является отрицанием  $X = Y$  (см. рис. 3).

Иногда при описании правил действий объекта возникает необходимость сослаться на текущий объект, т. е. на себя само-



го. Это можно сделать с помощью конструкции, называемой *локальным объектом*. Она имеет вид

**this A**

где A — идентификатор класса. Значением этой конструкции, которая может употребляться только в теле декларации класса A и декларациях его подклассов (см. 1.3), является ссылка на

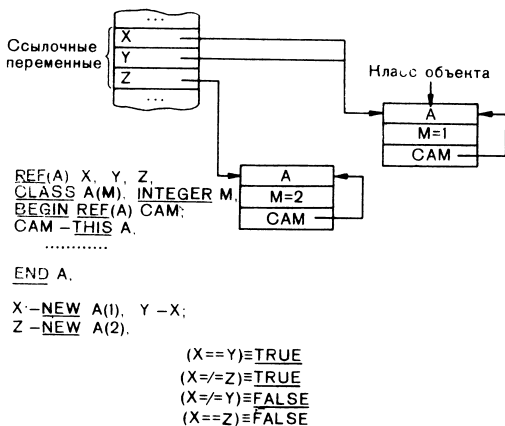


Рис. 3. Идентичность и неидентичность ссылочных переменных. Локальный объект.

объект класса A, исполняющий в данный момент свои правила действий. Пример использования локального объекта приведен на рис. 3.

Отметим, что ссылка на себя самого не предохраняет объект от уничтожения (освобождения занимаемой им памяти), если в программе не осталось других ссылок на этот объект. Например, после исполнения оператора

**Z:—none;**

(см. рис. 3) объект, на который до этого ссылалась переменная Z, становится недоступным (если, конечно, он не является текущим при исполнении оператора **Z:—none**) и занимаемая им память может быть использована для других целей.

### 1.3. Иерархическое описание классов объектов

Важной особенностью языка симула-67 является возможность структурированного описания сложных объектов в виде древовидных иерархий деклараций классов. В декларациях клас-

сов, расположенных на нижних уровнях иерархии, можно использовать все средства, определенные в декларациях вышестоящих уровней, и создавать на их основе новые классы объектов, которые наряду со всеми свойствами объектов верхних классов обладают еще рядом дополнительных, специфических свойств. Это позволяет адекватно и в удобной форме отображать иерархически построенные понятия и иерархические описания сложных объектов и систем, широко распространенные в науке и технике.

В имитационном моделировании применение постепенно наращиваемых иерархий деклараций классов наиболее удобно при нисходящей технологии разработки моделей, отражающей процесс все большей детализации представлений о моделируемой системе при переходе от одного этапа ее исследования или проектирования к другому. С помощью деклараций классов можно описывать типовые модели объектов, характерных для некоторой предметной области или класса систем. Эти типовые модели отображают основные свойства объектов и фиксируют опыт моделирования, накопленный их создателями. При построении конкретных моделей типовыми описаниями можно пользоваться как готовыми строительными блоками, которые, однако, при необходимости могут быть легко наделены дополнительными свойствами, отражающими специфику данной модели. Средства автономной компиляции деклараций классов, реализованные в трансляторах с языка симула-67 для БЭСМ-6 и ЕС ЭВМ (см. 4.2), позволяют хранить типовые модели в виде библиотек программ в архивах вычислительной системы и эффективно использовать их на всех этапах моделирования.

Рассмотрим имеющиеся в языке симула-67 средства отображения иерархических систем понятий на примере описания поведения шахматных фигур, делающих попытку встать на одну из свободных клеток шахматной доски, доступных за один ход из заданной начальной клетки. Решение этой задачи можно получить, описав ряд не связанных между собой деклараций классов, построенных аналогично декларации класса ПЕШКА (пример 1.2), каждая из которых полностью описывает поведение соответствующей фигуры. Эти декларации имели бы структуру, представленную в примере 1.6.

Пример 1.6.

```
class <НАЗВАНИЕ ФИГУРЫ> (НВ, НГ, БЕЛАЯ);  
  integer НВ, НГ; boolean БЕЛАЯ;  
  Н: begin integer ТВ, ТГ;  
    ТВ:=НВ; ТГ:=НГ;  
    if СВОБОДНА (ТВ, ТГ) then goto ОСТАНОВКА;
```

⟨операторы, описывающие поиск свободной клетки, доступной из начальной клетки с координатами (НВ, НГ) за один ход определяемой фигуры⟩; К: ОСТАНОВКА: ЗАНЯТЬ (ТВ, ТГ);  
**goto** КОНЕЦ; ВСТАТЬ НЕКУДА: ТВ:=ТГ:=0; КОНЕЦ;  
**end** декларации класса;

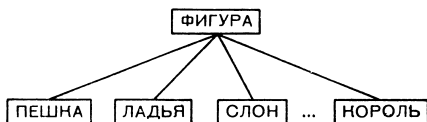
Метками Н и К здесь обозначены части деклараций классов, описывающие одинаковые действия, выполняемые любой фигурой в начальной (Н) и конечной (К) стадиях своего функционирования.

Как видно из примера 1.6, декларации классов для различных фигур отличаются по сути лишь операторами, описывающими выполнение хода, поскольку остальные действия и атрибуты одинаковы для всех шахматных фигур. Это позволяет выделить общие части деклараций классов, описывающих различные фигуры, в отдельную декларацию, задающую класс ФИГУРА, который имеет атрибуты и правила действий общие для всех фигур (см. пример 1.7).

Пример 1.7.

```
class ФИГУРА (НВ, НГ, БЕЛАЯ);
  integer НВ, НГ; boolean БЕЛАЯ;
begin integer ТВ, ТГ;
  ТВ:=НВ; ТГ:=НГ;
  if СВОБОДНА (ТВ, ТГ) then goto ОСТАНОВКА; inner;
  ОСТАНОВКА: ЗАНЯТЬ (ТВ, ТГ); goto КОНЕЦ;
  ВСТАТЬ НЕКУДА: ТВ:=ТГ:=0; КОНЕЦ;
end фигура;
```

В приведенной декларации символ **inner** представляет дополнительные действия, задаваемые в декларациях классов, описывающих поведение любых фигур. Теперь эти декларации можно определить на основе класса ФИГУРА и задавать в них только дополнительные, характерные лишь для данного вида фигур признаки (атрибуты и правила действия). Определенные таким образом классы называются подклассами класса ФИГУРА и образуют вместе с ним иерархию классов, структуру которой можно представить в виде следующего дерева:



Класс ФИГУРА называют префикс-классом (надклассом) для классов ПЕШКА, ЛАДЬЯ и т. д.

В симула-программе тот факт, что класс ПЕШКА является подклассом класса ФИГУРА, отображается употреблением имени ФИГУРА в качестве префикса к декларации класса ПЕШКА, которая теперь примет вид, показанный в примере 1.8.

Пример 1.8.

```
ФИГУРА class ПЕШКА;  
begin ТГ:=ТГ+(if БЕЛАЯ then 1 else - 1);  
if (ТГ > 8 or ТГ < 1) or not СВОБОДНА (ТВ, ТГ)  
then goto ВСТАТЬ НЕКУДА;  
end ПЕШКИ;
```

Вся иерархия классов, описывающих шахматные фигуры, задается набором деклараций

```
class ФИГУРА . . . . . ;  
ФИГУРА class ПЕШКА . . . . . ;  
ФИГУРА class ЛАДЬЯ . . . . . ;  
ФИГУРА class СЛОН . . . . . ;  
. . . . .  
ФИГУРА class КОРОЛЬ . . . . . ;
```

где каждый из подклассов, в свою очередь, может выступать в роли надкласса, что позволяет наглядно отображать системы понятий с древовидной структурой.

Работа объектов, принадлежащих к классу, определяемому декларацией с префиксом, происходит в соответствии с эквивалентной декларацией без префикса, получаемой путем объединения атрибутов и правил действий, которые заданы в декларациях класса и надкласса. Этот процесс объединения называется *сочленением* деклараций классов и выполняется в соответствии с определенными в языке правилами — алгоритмом сочленения. В процессе сочленения список формальных параметров декларации класса дополняется справа списком параметров декларации подкласса; аналогичным образом объединяются их совокупности спецификаций, а также описания внешних блоков тел сочленяемых деклараций. При объединении правил действий символ *inner* в теле класса заменяется операторами правил действий, заданными в декларации подкласса.

Применение алгоритма сочленения к декларациям *class* ФИГУРА . . . . . ; (пример 1.7) и *ФИГУРА class* ПЕШКА . . . . . ; (пример 1.8) даст эквивалентную беспрефиксную декларацию для класса ПЕШКА, которая будет полностью совпадать с декларацией, приведенной в примере 1.2 (см. 1.1).

Проиллюстрируем работу алгоритма сочленения на примере более сложной иерархии классов, структура которой задана деревом, изображенным на рис. 4, а.

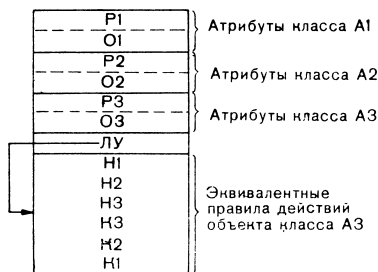
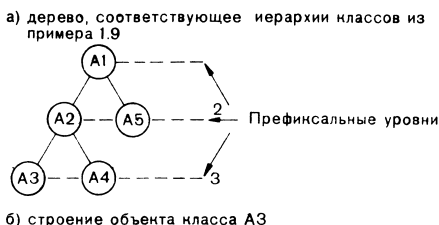


Рис. 4. Строение объекта, принадлежащего к классу с префиксом.

Декларации классов этой иерархии зададим схематически.  
Пример 1.9.

```
class A1(P1); C(P1); {O1; H1; inner; K1};
A1 class A2(P2); C(P2); {O2; H2; inner; K2};
A2 class A3(P3); C(P3); {O3; H3; inner; K3};
A2 class A4(P4); C(P4); {O4; H4; inner; K4};
A1 class A5(P5); C(P5); {O5; H5; inner; K5};
```

(скобки { , } соответствуют символам **begin** и **end**).

В примере 1.9 через  $P_k$  ( $k = 1, 2, \dots, 5$ ) обозначены атрибуты-параметры классов  $A_k$ , через  $C(P_k)$  — их спецификации, через  $O_k$  — описания самого внешнего блока тела  $A_k$ , через  $H_k$  и  $K_k$  соответственно операторы, задающие начальные и конечные действия объектов класса  $A_k$ .

Введем некоторые термины языка симула-67, связанные с иерархиями деклараций классов.

*Префиксальным уровнем* декларации класса и объектов, принадлежащих к этому классу, называется номер яруса соответствующей ей вершины в дереве иерархии.

*Цепочкой префиксов* некоторой декларации класса называется совокупность классов, лежащих на пути из корня дерева иерархии в вершину, соответствующую этой декларации.

Формирование эквивалентной декларации для некоторого класса  $A_k$  производится путем пошагового сочленения деклараций, лежащих на пути из корня дерева иерархии в вершину, соответствующую  $A_k$ , в порядке возрастания префиксального уровня. Например, эквивалентная декларация для класса  $A_2$  из примера 1.9 получается сочленением декларации `class A1(P1)`. . . .; и декларации `A1 class A2`. . . .; и имеет вид

```
class A2(P1, P2); C(P1); C(P2); {O1; O2; H1; H2; inner;  
K2; K1};
```

Символ `inner` при работе объектов класса  $A_2$  эквивалентен пустому оператору и сохранен в тексте декларации для дальнейшего сочленения. На основе полученной эквивалентной декларации для  $A_2$  и декларации `A2 class A3`. . . .; может быть получена эквивалентная декларация для  $A_3$ :

```
class A3(P1, P2, P3); C(P1); C(P2); C(P3);  
{O1; O2; O3; H1; H2; H3; inner; K3; K2; K1};
```

На рис. 4, б изображена структура объекта класса  $A_3$ , соответствующая приведенной эквивалентной декларации.

Аналогичный вид будут иметь и эквивалентные декларации для классов  $A_4$  и  $A_5$ . Символ `inner` в декларациях классов может быть опущен. Тогда считается, что он непосредственно предшествует завершающему символу `end` декларации, т. е. конечные действия в этом случае представляются пустым оператором.

Следует отметить, что появление операторов правил действий некоторого класса в эквивалентных декларациях его подклассов еще не означает многократного повторения соответствующих им участков в машинной программе, которую составляет компилятор по исходной симула-программе. Сочленение деклараций классов может быть реализовано такой организацией передач управления между участками машинной программы для правил действий сочленяемых деклараций, которая обеспечивает их выполнение в последовательности, задаваемой эквивалентной декларацией. Такой способ реализации сочленения используется в компиляторах для БЭСМ-6 и ЕС ЭВМ. Это позволяет экономить память, занимаемую программой в ЭВМ.

В языке симула-67 префиксами могут обладать не только декларации классов, но и блоки. Однако в роли префикса к блоку используется имя класса вместе с совокупностью фактических параметров, если, конечно, в соответствующей декларации

имеются формальные параметры. В блоке с префиксом можно пользоваться всеми атрибутами, определенными в декларации класса, указанного в префиксе, и в декларациях его надклассов. Пусть, например, декларация класса АВТОПАРК содержит описания объектов и алгоритмов, характерных для задач моделирования автохозяйств:

```
class АВТОПАРК (КОЛМАШ); integer КОЛМАШ;  
  begin ref (АВТОМОБИЛЬ) array МАШИНЫ [1 : КОЛМАШ];  
    class АВТОМОБИЛЬ...;  
    procedure РЕМОНТ...; Sн; inner; Sк  
  end АВТОПАРК;
```

Через S<sub>н</sub> и S<sub>к</sub> обозначены стандартные начальные и конечные действия, которые необходимы в любой модели автохозяйства. Имея декларацию класса АВТОПАРК, конкретную модель автохозяйства, состоящего из 250 машин, можно задать в виде блока с префиксом:

```
АВТОПАРК (250) begin АВТОМОБИЛЬ class ЗИЛ130...;  
  procedure ЗАПРАВКА...; S;  
end;
```

(через S обозначены операторы данной, конкретной, модели). При входе управления в блок с префиксом механизм сочленения обеспечивает выполнение операторов S в обрамлении операторов S<sub>н</sub> и S<sub>к</sub>, т. е. в последовательности S<sub>н</sub>; S; S<sub>к</sub>.

Блоки с префиксами играют важную роль в организации квазипараллельного взаимодействия объектов (см. 1.6) и при построении средствами языка симула-67 пакетов прикладных программ (см. главу 5).

Введение аппарата иерархий деклараций классов требует уточнения правил использования таких средств как квалификация переменных, генераторы объектов, присваивание ссылок.

При создании объектов, принадлежащих классу, декларация которого имеет префикс, в соответствующем генераторе объектов необходимо указать фактические значения для всех формальных параметров, присутствующих в эквивалентной декларации. Например, создание объекта класса АЗ (пример 1.9) должно задаваться генератором вида

```
new АЗ(F1, F2, F3)
```

где через F1, F2, F3 обозначены фактические параметры, соответствующие формальным параметрам P1, P2, P3, заданным в декларациях классов А1, А2, АЗ.

Ссылочная переменная или процедура-функция типа **ref** (<квалификация>) могут указывать не только на объекты клас-

са, который указан в их квалификации, но и на объекты любого из его подклассов, т. е. классов, расположенных в вершинах поддерева дерева иерархии деклараций, исходящего из вершины, соответствующей квалифицирующему классу. Рассмотрим в качестве примера следующие описания ссылочных переменных:

**ref** (A1) E1; **ref** (A2) E2; **ref** (A3) E3;

где A1, A2, A3 — имена классов из иерархии, представленной в примере 1.9. Переменная E1 может ссылаться на объекты любого из классов, составляющих эту иерархию, в то время как переменная E2 может указывать на объекты классов A2, A3, A4, а переменная E3 — лишь на объекты класса A3. Указанные ограничения на область возможных значений ссылочных переменных соблюдаются при выполнении любых операций присваивания ссылок. В примере 1.10 даны корректные последовательности операторов.

Пример 1.10.

а) E1:—**new** A4(...); E2:—E1;

б) E2:—**new** A2(...); E1:—E2;

Однако оператор

E2:—**new** A5(...);

является незаконным, причем ошибка в обозначении объекта выявляется при трансляции. Некорректна и последовательность операторов

E1:—**new** A1(...); E2:—E1;

хотя ошибка в данном случае будет обнаружена лишь во время исполнения оператора E2:—E1, поскольку он может выполнять и вполне законную операцию, если E1 ссылается на объект класса A2 или одного из его подклассов (см. пример 1.10, а).

Рассмотрим некоторые вопросы доступности атрибутов в иерархиях деклараций классов. В теле декларации класса можно непосредственно, т. е. по их именам, использовать все атрибуты, заданные в декларациях, составляющих цепочку префиксов этого класса. Атрибуты подкласса, напротив, из декларации надкласса непосредственно недоступны, хотя обратиться к ним можно с помощью дистанционных идентификаторов и виртуальных атрибутов (см. 1.4). Взаимно недоступны (непосредственно) и атрибуты деклараций, лежащих на разных ветвях в дереве иерархии.

В том случае, если в декларациях классов, принадлежащих одной цепочке префиксов, определены одноименные атрибуты, то при употреблении их имен в некоторой декларации обраще-



ние будет происходить к тому атрибуту, который определен в ее цепочке префиксов, причем в декларации с максимальным префиксальным уровнем.

Рассмотрим в качестве иллюстрации этих правил декларации, приведенные в примере 1.11.

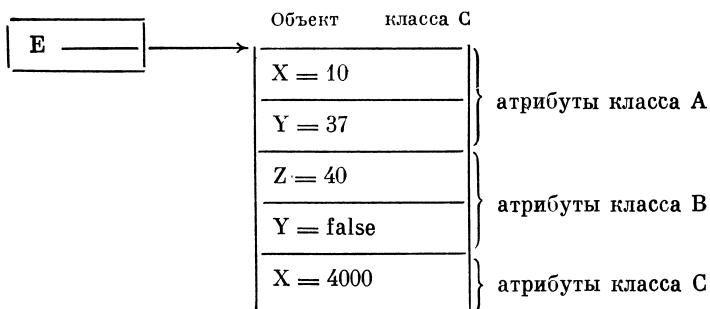
Пример 1.11.

```
class A(X, Y); real X, Y; {Y:=Y+X+7};
A class B(Z); integer Z; {boolean Y; Z:=X+Z; Y:=false};
B class C(X); integer X; {X:=if Y then 2*X else 100*X};
```

После исполнения оператора

```
E:—new C(10, 20, 30, 40);
```

атрибуты объекта E будут иметь следующие значения:



Над обозначениями объектов и именами классов в языке симула-67 определены два отношения, позволяющие определять принадлежность объекта к некоторому классу или его подклассу. Отношение  $X \text{ is } A$ , где  $X$  — объектное выражение (например, ссылочная переменная), а  $A$  — идентификатор класса, истинно, если объект, обозначаемый выражением  $X$ , принадлежит к классу  $A$ . Отношение  $X \text{ in } A$  истинно, если объект, на который указывает выражение  $X$ , принадлежит классу  $A$  или одному из его подклассов.

Например, после исполнения оператора (см. пример 1.11)

```
E:—new C(...);
```

отношения  $E \text{ is } C$ ,  $E \text{ in } A$ ,  $E \text{ in } B$  истинны, а после исполнения оператора

```
K:—new B(...);
```

где  $K$  описана как  $\text{ref } (B) K$ ; отношение  $K \text{ in } A$  — истинно, но  $K \text{ in } C$  — ложно.

## 1.4. Средства доступа к атрибутам объектов

1.4.1. Дистанционные идентификаторы. При выполнении операторов правил действий объекта, заданных в соответствующей ему декларации класса, обращение к его атрибутам производится непосредственно по их именам. Если декларация класса имеет префикс, то в ее операторах непосредственно доступны атрибуты всех надклассов.

Доступ к атрибутам других объектов выполняется с помощью дистанционных идентификаторов, которые содержат обозначение объекта и имя атрибута, к которому производится обращение.

В задачах имитационного моделирования использование дистанционных идентификаторов позволяет наглядно и эффективно описывать информационное взаимодействие объектов моделируемых систем.

Дистанционный идентификатор записывается в виде конструкции

$E.X$ ,

где  $E$  — обозначение объекта (простое объектное выражение), а  $X$  — идентификатор одного из атрибутов этого объекта. Если значением  $E$  является *none*, т. е.  $E$  не указывает ни на какой объект, то использование идентификатора  $E.X$  незаконно и вызовет сообщение об ошибке при исполнении симула-программы.

При определении атрибута  $X$  объекта  $E$ , к которому производится обращение посредством дистанционного идентификатора  $E.X$ , важную роль играет квалификация объектного выражения  $E$ . Если объектное выражение имеет вид переменной (простой или с индексами) или указателя функции, то его квалификацией считается квалификация, указанная в описании этой переменной (массива, процедуры-функции). Квалификацией генератора объектов или локального объекта является идентификатор класса, записанный после *new* или *this*.

Пусть  $C$  — квалификация объектного выражения  $E$ . В этом случае с помощью конструкции  $E.X$  можно обращаться только к тем атрибутам объекта, которые заданы в декларации класса  $C$  или декларациях его надклассов. Рассмотрим, например, следующий фрагмент симула-программы.

Пример 1.12.

```
ref (A) XA; ref (B) XB; ref (C) XC;  
class A (X); real X; begin ... end;  
A class B(Y); ref (A) Y; begin ... end;
```

```

B class C; begin integer I; array M [1:5]; I:=8; ...end;
XA:—new A(5); XB:—new B(10, XA);
XC:—new C(20, new A(30));

```

В результате выполнения последних трех операторов будут созданы 4 объекта, которые схематически изображены на рис. 5.

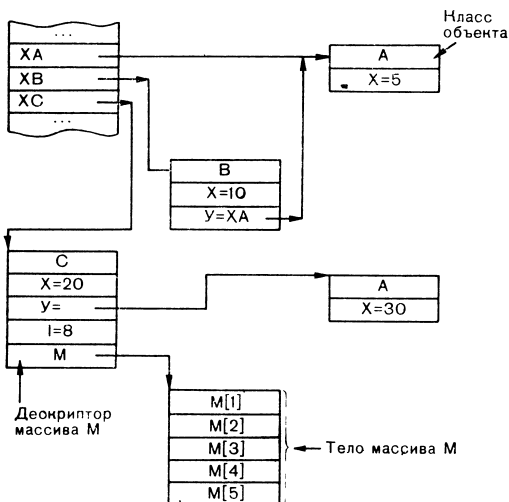


Рис. 5. Структура данных, соответствующая фрагменту программы из примера 1.12.

Обращаться к атрибутам созданных объектов можно с помощью дистанционных идентификаторов через переменные XA, XB, XC из любой точки программы, на которую распространяется действие их описаний. Например, оператор  $XA.X := XA.X + 100$ ; увеличит на 100 значение атрибута X у объекта XA. То же самое действие выполнит оператор  $XB.Y.X := XB.Y.X + 100$ ; После выполнения оператора  $XB.X := XC.X + XC.Y.X$ ; атрибут X объекта XB будет иметь значение 50.

Выбор атрибута объекта в соответствии с квалификацией ссылки в дистанционном идентификаторе служит средством защиты данных и исключает возможность обращения к несуществующим атрибутам. Чем шире область возможных значений ссылочной переменной, тем уже множество атрибутов, к которым можно обратиться посредством употребления этой переменной в дистанционном идентификаторе. Например, переменная, квалифицированная корневым классом некоторой иерархии, может ссылаться на объекты любого класса из этой иерархии, однако, на какой бы объект она ни ссылалась, обратиться

через нее можно только к атрибутам, определенным в корневой декларации (см. рис. 6).

Дистанционные идентификаторы, использующие локальный объект, позволяют в теле декларации подкласса обращаться к

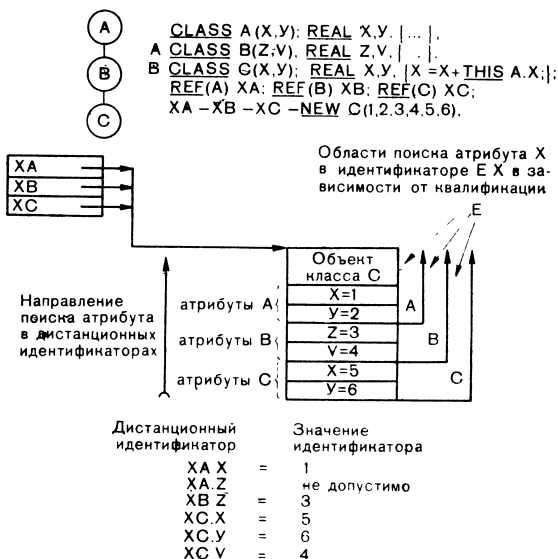


Рис. 6. Выбор атрибутов в дистанционных идентификаторах.

тем атрибутам надкласса, которые непосредственно недоступны из-за конфликтов наименований. Например, оператор

$X := X + \text{this A.X};$

в теле декларации класса C увеличит значение атрибута X, определенного в классе C, на величину, равную значению атрибута X, заданного в декларации класса A (см. рис. 6).

В тех случаях, когда заведомо известно, что переменная, квалифицированная надклассом, ссылается на объект, принадлежащий к подклассу, обратиться посредством этой переменной к атрибутам объекта, определенным в декларации подкласса, можно путем задания так называемой *оперативной квалификации*, которая имеет вид

*qua C,*

где C — идентификатор класса, и пишется после имени ссылочной переменной (в общем случае — после объектного выражения). Ее употребление сигнализирует о том, что предшествующее объектное выражение ссылается на объект класса C или

его подкласса, и разрешает обращение к его атрибутам, определенным в декларации класса С или декларациях его надклассов. Например, переменная ХА (см. пример 1.12) имеет более общую квалификацию, чем ХВ и ХС, так как классы В и С являются подклассами класса А, поэтому ХА может ссылаться и на объекты, обозначенные переменными ХВ и ХС. Например, оператор ХА:—ХС заставит ХА указывать на тот же объект, что и ХС, однако, обратиться через ХА с помощью дистанционных идентификаторов ХА.У и ХА.І к атрибутам У и І объекта ХС нельзя, так как эти атрибуты отсутствуют в декларации класса А, служащего квалификацией ХА. Однако, употребив оперативную квалификацию **qua** С к этим атрибутам можно обратиться с помощью дистанционных идентификаторов ХА **qua** С.У и ХА **qua** С.І соответственно.

С помощью дистанционных идентификаторов можно обращаться к атрибутам-массивам и атрибутам-процедурам. Если вернуться к примеру 1.12, то следующий фрагмент программы присвоит каждому элементу массива М, служащего атрибутом объекта ХС, значение, равное его номеру:

```
integer I;
for I:=1 step 1 until 5 do ХС.М [I]:=I;
```

С помощью атрибутов-процедур удобно задавать такие операции над объектами, которые могут инициироваться как самим объектом, так и извне его, т. е. другими объектами. Таким образом можно описывать, в частности, абстрактные типы данных с системой операций над ними. Рассмотрим в качестве простого примера описание класса ТОЧКА. (см. пример 1.13), задающего точки, которые располагаются на плоскости, причем каждая из них характеризуется координатами (X, Y) и обладает способностью двигаться в вертикальном и горизонтальном направлениях.

Пример 1.13.

```
class ТОЧКА (X, Y); real X, Y;
begin
  procedure ВПРАВО (S); real S; X:=X+S;
  procedure ВВЕРХ (H); real H; Y:=Y+H;
end;
```

Разместить на плоскости точки А и В с координатами (1, 2) и (3, 4) соответственно можно с помощью следующего фрагмента программы:

```
ref (ТОЧКА) А, В;
А:—new ТОЧКА (1, 2);
В:—new ТОЧКА (3, 4);
```

Сдвинуть точку А вправо на 5 единиц можно посредством исполнения оператора

А.ВПРАВО (5);

а оператор

В.ВВЕРХ (—10);

сдвинет точку В вниз на 10 единиц. Если нужно поднять точку А на столько единиц, каково значение суммы координат точки В, то достаточно написать оператор

А.ВВЕРХ (В.Х+В.У);

В примере 1.14 приведена декларация класса ВЕКТОР, построенная на основе декларации класса ТОЧКА, которая определяет двумерные вектора с началом в точке (0, 0) и операцию сложения векторов.

Пример 1.14.

ТОЧКА class ВЕКТОР;

begin

ref (ВЕКТОР) procedure ПЛЮС (P); ref (ВЕКТОР) P;

ПЛЮС:—new ВЕКТОР (X+P.X, Y+P.Y);

end класса ВЕКТОР;

Следующий фрагмент программы описывает генерацию векторов  $\vec{X}$  и  $\vec{Y}$ , получение их суммы  $\vec{S} = \vec{X} + \vec{Y}$ , сложение трех векторов, сдвиг конца полученного вектора на 15 единиц вправо:

ref (ВЕКТОР) X, Y, S;

X:—new ВЕКТОР (5, 15); Y:—new ВЕКТОР (10, 20);

S:—X.ПЛЮС (Y); X:—X.ПЛЮС (S).ПЛЮС (Y); X.ВПРАВО (15);

1.4.2. Присоединяющие операторы. Еще одним средством доступа к атрибутам объектов служат присоединяющие операторы, которые удобно использовать там, где часто приходится обращаться к атрибутам одного и того же объекта. Наиболее употребительная форма присоединяющего оператора имеет вид

inspect E do S;

где E — объектное выражение, а S — оператор, в котором можно обращаться к атрибутам объекта E непосредственно по их именам. Объект E принято называть присоединенным объектом, а S — присоединяющим блоком. В качестве примера использования присоединяющих операторов приведем оператор, увеличивающий значения координат точки А [I] (пример 1.13 из раз-

дела 1.4.1) на величину, равную текущему расстоянию этой точки от начала координат.

**Пример 1.15.**

```
ref (ТОЧКА) array A [1 : 10];  
inspect A [I] do  
  begin real P;  
    P:=sqrt(X**2+Y**2); X:=X+P; Y:=Y+P;  
  end;
```

Те же самые действия в примере 1.16 описаны без применения присоединения, с использованием дистанционных идентификаторов. При этом приходится многократно пользоваться конструкциями  $A[I].X$  и  $A[I].Y$  вместо  $X$  и  $Y$  в примере 1.15. Это может оказаться неудобным, особенно если в роли  $A$  выступает, в свою очередь, дистанционный идентификатор.

**Пример 1.16.**

```
begin real P;  
  P:=sqrt (A[I].X**2+A[I].Y**2);  
  A[I].X:=A[I].X+P; A[I].Y:=A[I].Y+P;  
end;
```

Если при присоединении к объекту нужно выполнять разные действия в зависимости от того, к какому классу принадлежит этот объект, то применяется другая форма присоединяющего оператора:

```
inspect E when A1 do S1  
  when A2 do S2  
  . . . . .  
  when AK do Sk;
```

где  $A_1, A_2, \dots, A_K$  — идентификаторы классов, а  $S_1, S_2, \dots, S_k$  — операторы, причем в  $S_i$  ( $i = 1, 2, 3, \dots, k$ ) возможно непосредственное использование атрибутов объекта  $E$ , заданных в декларации класса  $A_i$ . Если  $E$  принадлежит классу  $A_i$  (или одному из его подклассов), то исполняется оператор  $S_i$ , после чего управление передается на оператор, следующий за всем присоединяющим оператором, т. е. операторы  $S_m$  ( $m \neq i$ ) не выполняются.

**1.4.3. Виртуальные атрибуты.** Некоторые из атрибутов-процедур (а также переключателей и меток), определенных в декларации класса, могут быть объявлены в ней *виртуальными*. Для виртуальных атрибутов характерно особое поведение в процессе формирования эквивалентной декларации: описание виртуального атрибута, расположенное в декларации класса с префиксальным уровнем  $K$ , замещается описанием с

таким же идентификатором, но расположенным в декларациях с большим, чем К уровнем (в декларациях подклассов). Таким образом, объявление атрибута виртуальным в некоторой декларации класса С позволяет переопределять его в декларациях подклассов и обращаться посредством этого атрибута к другим атрибутам подклассов, которые из С непосредственно недоступны. Рассмотрим в качестве примера следующую иерархию классов.

**Пример 1.17.**

```
class A(X); real X; virtual: real procedure P;
  begin real Y, Z;
    real procedure P; P:=Y+Z; X:=P;
  end;
A class B;
  begin real K, M; real procedure P; P:=K+M; end;
B class C;
  begin real L, Q; real procedure P; P:=L+Q; end;
```

При работе объектов класса А оператор  $X:=P$  присвоит атрибуту X сумму значений атрибутов Y и Z посредством обращения к процедуре-функции P, описанной в декларации А. Однако, когда этот же оператор будет исполняться в составе правил действий объектов класса В, роль P будет играть уже процедура, описанная в декларации В, поскольку атрибут P объявлен виртуальным в декларации А, и X примет значение, равное сумме значений атрибутов K и M. Для объектов класса С оператор  $X:=P$  будет эквивалентен оператору  $X:=L+Q$ .

## 1.5. Организация упорядоченных множеств объектов

В практике программирования часто возникает необходимость объединения объектов в упорядоченные множества. В частности, при создании имитационных моделей нужно отображать разного рода очереди, а также структурные взаимосвязи объектов в моделируемых системах. Использование в качестве атрибутов объектов ссылок на другие объекты позволяет организовывать из них структуры данных любого вида.

Рассмотрим средства организации и обработки упорядоченных множеств объектов, имеющиеся в системном классе `simset` и их применение для отображения очередей (1.5.1), а также для построения из отдельных объектов иерархических (1.5.2) и сетевых (1.5.3) структур данных.

**1.5.1. Класс `simset`.** Средства, определенные в системном классе `simset`, позволяют представлять и обрабатывать упорядоченные множества, организованные в виде циклических



двунаправленных списков входящих в них объектов. Такие множества в литературе по языку симула-67 принято называть *наборами*. Мы достаточно подробно рассмотрим строение класса *simset*, поскольку он служит хорошим примером применения языка симула-67 для разработки специализированного программного обеспечения, ориентированного на некоторую предметную область, в данном случае — на создание и обработку упорядоченных множеств объектов. Кроме того, класс *simset* служит основой для построения другого системного класса — *simulation*, в котором содержатся средства для описания имитационных моделей дискретных систем.

Общая структура класса *simset* описывается следующей декларацией:

```
class simset;
  begin class linkage;
    begin ref (linkage) SUCC, PREDD;
      ref (link) procedure suc; ...;
      ref (link) procedure pred; ...;
    end LINKAGE;
  linkage class link;
    begin procedure out; ...;
      procedure follow (X); ref (linkage) X; ...;
      procedure precede (X); ref (linkage) X; ...;
      procedure into (S); ref (head) S; ...;
    end LINK;
  linkage class head;
    begin ref (link) procedure first; ...;
      ref (link) procedure last; ...;
      boolean procedure empty; ...;
      integer procedure cardinal; ...;
      procedure clear; ...;
      SUCC:—PREDD:—this linkage;
    end head;
  end simset;
```

Общим свойством всех объектов, объединенных в множество с помощью циклического двунаправленного списка, является наличие ссылок на своего преемника и предшественника. Эти ссылки представляются атрибутами SUCC и PREDD, определенными в декларации класса *linkage*, служащего надклассом для классов *link* и *head*, в которых описаны дополнительные свойства, присущие членам наборов.

В классе *link* определены процедуры, позволяющие включать в набор и исключать из набора объекты этого класса и его подклассов: процедура *out* исключает объект, атрибутом которо-

го она является, из содержащего его набора; процедуры **follow** и **precede** дают возможность включить объект в набор непосредственно после или перед объектом **X**; процедура **into** вставляет объект в конец набора **S** (вслед за его последним элементом).

В роли начала и конца набора выступает объект класса **head**, называемый *головой набора*. Процедуры **first** и **last**, описанные в декларации класса **head**, дают доступ к крайним элементам набора (соответственно к первому и последнему), процедура **empty** дает значение **true**, если соответствующий набор пуст, т. е. содержит только голову набора. Значение процедуры **cardinal** равно числу членов набора, не считая его головы, а с помощью процедуры **clear** можно исключить из набора все его члены, сделав набор пустым.

Создается набор при генерации объекта класса **head**, который вначале является своим собственным преемником и предшественником: оператор

**SUCC:—PREDD:—this linkage;**

заносит в атрибуты **SUCC** и **PREDD** этого объекта ссылку на самого себя, являющуюся значением выражения **this linkage;**.

Важно отметить, что атрибуты **SUCC** и **PREDD** доступны для пользователя только через процедуры, определенные в классах **linkage**, **link**, **head**, что гарантирует корректное использование этих ссылок при работе с множествами.

Понятие набор, введенное в классе **simset**, проиллюстрировано на рис. 7.

Рассмотрим в качестве примеров описания нескольких вышеупомянутых процедур класса **simset** на языке симула-67.

**Пример 1.18.**

**a) ref (linkage) procedure suc;**

**suc:—if SUCC in link then SUCC else none;**

Эта процедура позволяет организовать просмотр набора по ссылкам на преемника. Когда будет достигнута голова набора, значением процедуры станет **none** (отношение **SUCC in link** примет значение **false** при попытке взять процедурой **suc** ссылку на объект, следующий за последним членом набора).

**б) procedure out;**

**if SUCC/=none then**

**begin**

**SUCC.PREDD:—PREDD; PREDD.SUCC:—SUCC;**

**SUCC:—PREDD:—none;**

**end out;**

```

в) integer procedure cardinal;
  begin integer I; ref (linkage) X; X:—this linkage;
    for X:—X.succ while X≠none do I:=I+1;
    cardinal:=I;
  end cardinal;

```

В следующем фрагменте симула-программы (пример 1.19) приведены примеры использования средств класса `simset` для

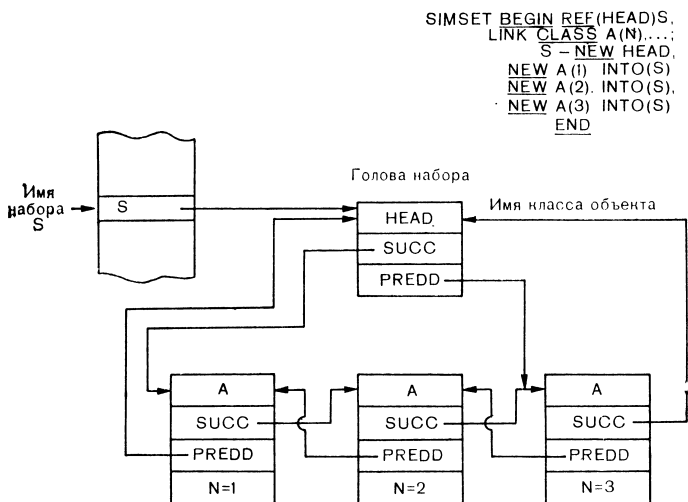


Рис. 7. Структура упорядоченных множеств в классе SIMSET.

имитации постановки двух автомобилей в очередь, отображаемую набором `СТОЯНКА`. Пользоваться средствами класса `simset` можно, употребив его идентификатор в качестве префикса к блоку или классу.

**Пример 1.19.**

```

simset begin
  link class АВТОМОБИЛЬ (...); ...;
  ref (head) СТОЯНКА; ref (АВТОМОБИЛЬ) A1, A2;
  СТОЯНКА:—new head;
  A1:—new АВТОМОБИЛЬ (...);
  A2:—new АВТОМОБИЛЬ (...);
  comment создали набор СТОЯНКА и автомобили A1
  и A2;
  A1.into (СТОЯНКА);
  A2.precede (A1);

```

```

comment включили A1 в очередь, а затем A2, постави-
ли перед A1;
СТОЯНКА. first. out;
comment автомобиль A2 покинул стоянку;
A1.out;...
comment_автомобиль A1 покинул стоянку;

. . .
K:=СТОЯНКА. cardinal;
comment значение K равно числу автомобилей в наборе
СТОЯНКА;

. . .
end блока;

```

Аппарат префиксов (1.3) позволяет дополнять, а при необходимости и переопределять средства, предоставляемые классом `simset`. Пусть, например, в процессе работы с упорядоченными множествами приходится часто вычислять количество

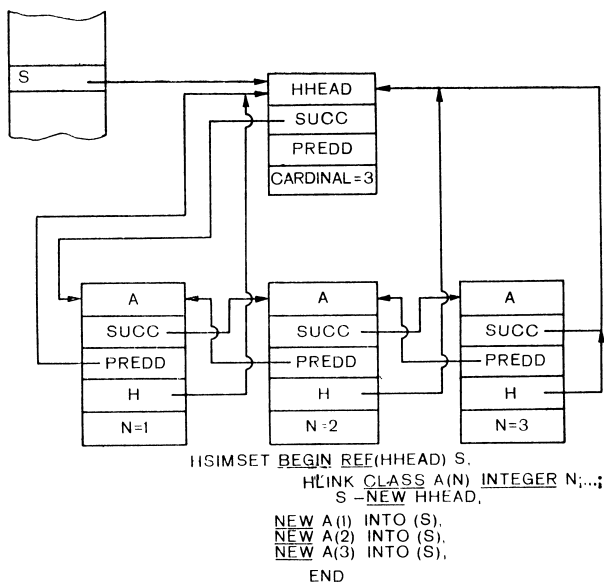


Рис. 8. Структура наборов, определяемых классом `HSIMSET`.

объектов, входящих в то или иное множество. Это удобно делать с помощью конструкции `S.cardinal`, представляющей обращение к процедуре-функции `cardinal`, которая является атрибутом объекта `S` класса `head` и определяет число членов набора путем их перебора (см. пример 1.18, в). Это может потребовать от-

носителю больших затрат времени, если наборы содержат много членов. Перебора членов набора можно избежать, если упорядоченное множество будет организовано так, что его голова содержит целое число, равное текущему количеству членов множества. Если этому атрибуту дать имя `CARDINAL`, то прежнее обращение `S. CARDINAL` будет давать число членов набора, но уже без их перебора. При этом, конечно, нужно корректировать значение атрибута `CARDINAL` в голове набора при вставлении и исключении его членов. Для эффективного выполнения этой корректировки в каждом объекте, который может быть членом набора, следует предусмотреть специальный атрибут, скажем `H`, содержащий ссылку на голову набора. Структура данных, соответствующая такому множеству, изображена на рис. 8. В примере 1.20 приводится декларация класса `HSIMSET`, обеспечивающего работу с множествами указанной структуры.

**Пример 1.20.**

```
simset class HSIMSET;
begin head class HHEAD;
  begin integer CARDINAL;
    CARDINAL:=0;
  end HHEAD;
link class HLINK;
  begin ref (HHEAD) H;
  comment теперь определим новые процедуры into и out,
дополнив действия, заданные в старых into и out, опи-
санных в классе link;
  procedure INTO (S); ref (HHEAD) S;
    begin this link. into (S);
    comment обратились к старой процедуре into;
    H:=S; S. CARDINAL:=S. CARDINAL+1;
    comment выполнили дополнительные действия, связан-
ные с вставлением объекта в список;
    end INTO;
  procedure OUT;
    begin this link. out;
    H. CARDINAL:=H. CARDINAL-1;
    comment уменьшили на 1 текущее число членов набора;
    H:=none;
    comment уничтожили ссылку на голову набора;
    end OUT;
    comment аналогичным образом могут быть переопреде-
лены и другие процедуры класса simset;
  end HLINK;
end HSIMSET;
```

Отметим, что при переходе на использование средств класса `HSIMSET` программа, в которой ранее применялись средства класса `simset`, практически не нуждается в изменениях: достаточно сменить квалификации у переменных, обозначающих наборы с `head` на `HHEAD`, заменить имя класса `head` на `HHEAD` в соответствующих генераторах объектов и вместо префикса `link` в декларациях классов, описывающих объекты, которые могут быть членами наборов, поставить префикс `HLINK` (см. рис. 8). В блоке с префиксом `HSIMSET` можно, конечно, пользоваться и всеми средствами класса `simset`, т. е. можно работать с наборами, имеющими старую структуру. Для этого у переменных, обозначающих старые наборы, нужно сохранить квалификацию `head`, а у деклараций классов, объекты которых могут быть их членами, оставить прежний префикс — `link`.

**1.5.2. Представление и обработка иерархических структур.** Рассмотрим применение наборов и процедур, определенных в классе `simset`, для отображения и обработки иерархических древовидных структур объектов. Узлы дерева удобно отображать с помощью объектов, которые можно включать в наборы, т. е. объектов, принадлежащих подклассу класса `link`. Ветви дерева, исходящие из некоторого узла, можно представить набором, членами которого являются объекты, соответствующие узлам следующего яруса, связанным с этим узлом. Дополнительная информация, характеризующая узлы дерева, может представляться значениями атрибутов соответствующих им объектов.

Таким образом, объекты, отображающие узлы дерева, могут быть описаны следующей декларацией класса.

**Пример 1.21.**

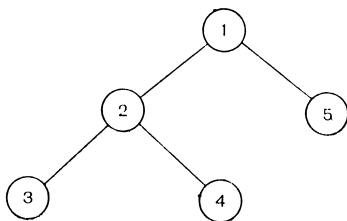
```
link class УЗЕЛ (НОМЕР); integer НОМЕР;  
  begin ref (head) ВЕТВИ; integer M;  
    ВЕТВИ:—new head;  
  end УЗЕЛ;
```

В данном примере предполагается, что каждый узел характеризуется дополнительно двумя целыми числами: порядковым номером (атрибут `НОМЕР`) и номером яруса, в котором он находится (атрибут `M`).

Опишем в виде рекурсивной процедуры один из возможных алгоритмов генерации дерева из  $K$  узлов. Алгоритм генерирует узлы дерева в порядке возрастания их номеров и включает каждый из них в набор `ВЕТВИ` соответствующего узла верхнего яруса.

Нумерация вершин дерева и их генерация производится в направлении «сверху — вниз» и «слева — направо». Будем счи-

тать, что количество ветвей для узла с номером  $I$  задано в  $I$ -м элементе массива КОЛВЕТВЕЙ  $[1:K]$ . Например, для генерации дерева следующего вида



(числа в узлах обозначают их порядковые номера). Элементы массива КОЛВЕТВЕЙ должны иметь такие значения: 2, 2, 0, 0, 0. Введем глобальные переменные КОРЕНЬ и ТН для обозначения корня дерева и текущего номера генерируемого узла. Фрагмент симула-программы, приведенный в примере 1.22, производит генерацию дерева, узлы которого отображаются объектами класса УЗЕЛ (см. пример 1.21).

**Пример 1.22.**

```

integer КОЛВЕТВЕЙ [1 : K]; integer ТН;
ref (УЗЕЛ) КОРЕНЬ;
procedure СВЯЗАТЬ (У); ref (УЗЕЛ) У;
comment процедура генерирует поддереву, исходящее из
узла У;
  begin integer КВ; КВ:=КОЛВЕТВЕЙ [У. НОМЕР];
  for КВ:=КВ-1 while КВ ≥ 0 do
    begin ref (УЗЕЛ) НОВЫЙ; ТН:=ТН+1;
      НОВЫЙ:=new УЗЕЛ (ТН);
      НОВЫЙ. into (У. ВЕТВИ); СВЯЗАТЬ (НОВЫЙ);
    end
  end СВЯЗАТЬ;
КОРЕНЬ:=new УЗЕЛ (1); ТН:=1; СВЯЗАТЬ (КОРЕНЬ);
  
```

В качестве примера обработки иерархических структур рассмотрим процедуру, выполняющую обход всех узлов дерева и помечающую каждый узел целым числом, равным номеру яруса этого узла. Будем считать, что узлы дерева представлены объектами класса УЗЕЛ (пример 1.21), и номер яруса нужно занести в атрибут М каждого объекта.

Опишем процедуру ЯРУС (К, L), которая выполняет указанную пометку для узлов поддерева с корнем К, расположенным на ярусе L.

### Пример 1.23.

```
procedure ЯРУС (K, L); ref (УЗЕЛ) K; integer L;  
begin ref (УЗЕЛ) T;  
  K. M:=L,  
  comment пометим узел K, после чего помечаем узлы  
  яруса (L+1), расположенные на ветвях исходящих из  
  K, просматривая набор ветви узла K;  
  T:=K. ВЕТВИ. first;  
  while T=/=none do  
    begin ЯРУС (T, L+1); T:=T. suc;  
    end пометки яруса (L+1);  
  end ЯРУС;
```

Имея процедуру ЯРУС, пометку узлов дерева, генерируемого программой из примера 1.22, можно выполнить с помощью оператора ЯРУС (КОРЕНЬ, 1).

1.5.3. Отображение сетевых структур. Рассмотрим один из возможных способов представления ориентированных графов с нагруженными дугами и узлами посредством совокупности объектов, связанных друг с другом взаимными ссылками. Для определенности будем рассматривать отображение городов и путей сообщения между ними.

Каждый город будем характеризовать двумя целыми числами: условным номером (Н) и количеством жителей (К), а дорогу из города в город — ее длиной (ДЛИНА) и пропускной способностью (ПС). Совокупность дорог, исходящих из города, можно представить набором ДОРОГИ, членами которого являются объекты класса ДОРОГА. Пункт назначения каждой дороги указывается в атрибуте КУДА, значением которого является ссылка на соответствующий объект класса ГОРОД.

Формальное описание объектов класса ГОРОД и ДОРОГА может быть дано с помощью следующих деклараций.

### Пример 1.24.

```
class ГОРОД (Н, К); integer Н, К;  
begin ref (head) ДОРОГИ;  
  ДОРОГИ:=new head;  
end ГОРОД;  
link class ДОРОГА (КУДА, ДЛИНА, ПС);  
  ref (ГОРОД) КУДА; real ДЛИНА, ПС; ;
```

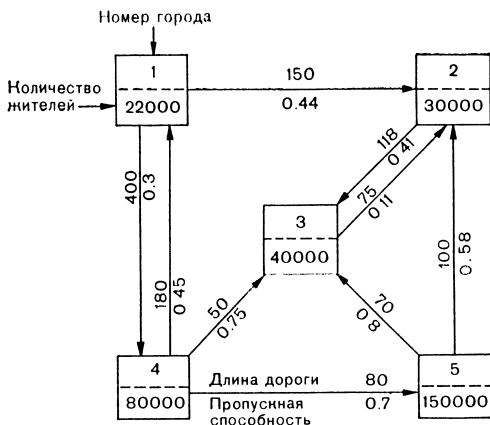
З а м е ч а н и е. Телом декларации класса ДОРОГА является пустой оператор, поскольку никаких действий его объектам мы не приписываем.

Фрагмент симула-программы, приведенный в примере 1.25, описывает формирование сети из N городов, связанных M дорогами. Предполагается, что данные о городах и дорогах распо-



ложены на системном устройстве ввода и передаются в программу посредством обращения к встроенным процедурам `inint` и `inreal`, выполняющим ввод целых и вещественных чисел соответственно.

Информация о количестве жителей в городах задается  $N$  целыми числами, а структура дорожной сети и данные о длине и пропускной способности каждой дороги определяются с помощью  $M$  четверок чисел: первые два числа указывают номера



Входные данные для программы в примере 1.25.

		N=5		M=9	
22000	30000	40000	80000	150000	
1	4	400	0.3		
1	2	150	0.44		
2	3	118	0.41		
3	2	75	0.11		
4	1	180	0.45		
4	3	50	0.75		
4	5	80	0.7		
5	3	70	0.8		
5	2	100	0.58		

Рис. 9. Ориентированный граф с нагруженными узлами и его числовое представление.

городов, которые связывает эта дорога, а следующие задают длину и пропускную способность. На рис. 9 изображена конкретная сеть дорог и ее представление с помощью ряда чисел, которое используется в качестве входных данных для программы из примера 1.25.

Пример 1.25.

`N:=inint; M:=inint;`

`simset begin ref (ГОРОД) array ГОРОДА [1:N]; integer I;`

`class ГОРОД (H, K); ...;`

`link class ДОРОГА (КУДА, ДЛИНА, ПС); ...;`

**for I:=1 step 1 until N do**

**ГОРОДА [I]:—new ГОРОД (I, inint);**

**comment** сгенерированы N объектов, отображающих узлы сети;

**for I:=1 step 1 until M do**

**begin integer A, B; real L, P;**

**A:=inint; B:=inint; L:=inreal; P:=inreal;**

**comment** ввели данные об одной дороге;

**new ДОРОГА (ГОРОДА[B], L, P). into (ГОРОДА [A]. ДОРОГИ);**

**comment** создали объект, отображающий дорогу из города A в город B длиной L и пропускной способностью P;

**end цикла формирования дорог;**

**comment** сеть готова для дальнейшей обработки;

. . .  
<использование и обработка сети>

. . .

**end;**

## 1.6. Динамическое взаимодействие объектов. Сопрограммы

Важным свойством объектов в языке симула-67 является их способность функционировать *квазипараллельно*, т. е. выполнять операторы, составляющие правила действия объекта за несколько активных фаз, в промежутках между которыми работают другие объекты. Такой механизм функционирования объектов лежит в основе средств имитационного моделирования (см. главу 2), где с его помощью отображается параллельная (одновременная) работа элементов моделируемых систем. Кроме того, возможность организации квазипараллельного исполнения компонентов программы оказывает существенное влияние на стиль программирования, позволяя (там, где это удобно) представлять решение задачи в виде взаимодействия ряда объектов, а не в виде последовательного вычислительного процесса.

В 1.6.1 неформально, на содержательном уровне, рассматриваются средства квазипараллельного исполнения, а в 1.6.2 приводится пример их применения для решения задачи о расстановке N ферзей на шахматной доске размером  $N \times N$  так, чтобы они не били друг друга.

1.6.1. Операторы **detach** и **resume**. Организация квазипараллельной работы объектов в языке симула-67 выполняется с помощью операторов **detach** (открепить) и **resume** (возобно-

вить), которые управляют чередованием активных фаз объектов. На основе этих операторов и средств иерархического описания классов объектов (см. 1.3) в системном классе *simulation* (глава 2) строятся средства имитационного моделирования дискретных систем. Таким образом, при построении имитационных моделей на языке симула-67 нет необходимости пользоваться операторами *detach* и *resume*, поскольку системный класс *simulation* предоставляет более мощные средства для отображения взаимодействия параллельно работающих объектов. Однако знание семантики базовых операторов *detach* и *resume* позволит пользователю создавать свои средства моделирования, ориентированные на имитацию конкретных классов систем.

Рассмотрим действия, выполняемые над объектами операторами *detach* и *resume*. Эти действия зависят от состояния, в котором находится объект. В языке симула-67 выделяют следующие состояния объекта: «прикреплен», «самостоятелен» («откреплен»), «завершен».

В прикрепленном состоянии объект находится с момента начала вычисления соответствующего генератора объектов до первого выполнения оператора *detach* в его правилах действий. Исполнение прикрепленным объектом оператора *detach* завершает вычисление генератора объектов. При этом управление возвращается в блок (объект, процедуру), содержащий генератор объекта, и в качестве результата его работы передается ссылка на созданный объект, который переходит в состояние «самостоятелен» (открепляется). Термины «открепляется», «прикреплен» отражают тот факт, что в процессе вычисления генератора объектов создаваемый объект считается частью объекта, содержащего его генератор, или, другими словами, прикреплен к нему.

При откреплении объекта его локальное управление оставливается на операторе, следующем за *detach*. С этой точки начнется исполнение правил действий объекта во время его следующей активной фазы.

Самостоятельное состояние объекта характеризуется тем, что он более не связан с породившим его объектом и может продолжить выполнение своих правил действий с того оператора, на который указывает его локальное управление. Задать возобновление работы самостоятельного объекта можно с помощью оператора

*resume* (X);

где X — объектное выражение, указывающее на объект, исполнение правил действий которого нужно возобновить.

При выполнении оператора `resume (X)` в правилах действий некоторого объекта  $Y$  происходит передача центрального управления от объекта  $Y$  к объекту  $X$ . При этом локальное управление объекта  $Y$  останавливается на операторе, следующем за `resume (X)`, и он перестает быть активным, а объект  $X$  возобновляет выполнение своих операторов с той точки, на которую указывало его локальное управление.

Вся симула-программа считается самостоятельным объектом, который получает управление от операционной системы. Этот объект принято называть *главной программой* (ГП). В процессе работы ГП порождает другие объекты, которые в процессе работы взаимодействуют между собой и с главной программой. Передача управления между объектами, отличными от ГП, осуществляется с помощью операторов `resume(X)`, в аргументах которых указывается активизируемый объект. Возобновить таким же образом работу главной программы нельзя, так как на нее нельзя ссылаться с помощью объектных выражений (более подробно об этом см. [10]). В связи с этим для передачи управления от самостоятельного объекта главной программе служит оператор `detach`, исполнение которого самостоятельным объектом эквивалентно исполнению оператора `resume` со ссылкой на ГП в качестве аргумента.

Если в симула-программе имеется блок с префиксом, то он играет роль главной программы для объектов, чьи декларации классов содержатся в данном блоке. В литературе по языку симула-67 блоки с префиксами называются также *квазипараллельными системами* (КП-системами) [10]. В принципе, симула-программа может содержать несколько блоков с префиксами, которые образуют дерево КП-систем. Корнем этого дерева служит самый внешний блок симула-программы, по определению являющийся КП-системой. Мы не будем более подробно рассматривать связанные с этим вопросы, поскольку для дальнейшего изучения программирования на языке симула-67 достаточно изложенных выше сведений. В описаниях эталонного языка, например [10], можно найти строгие формальные определения всех обсуждавшихся здесь понятий и основанное на них описание семантики операторов `detach` и `resume`.

Рассмотрим пример организации квазипараллельной работы объектов с помощью операторов `detach` и `resume`. Пусть требуется обеспечить следующую схему взаимодействия: два объекта (ОА и ОВ) во время своих активных фаз изменяют значение глобальной переменной  $M$ , причем делают это, поочередно передавая друг другу управление до тех пор, пока  $M \leq 1000$ . При  $M > 1000$  нужно закончить работу, напечатав сообщение КОНЕЦ РАБОТЫ. Программа, приведенная в при-

мере 1.26, обеспечивает требуемое взаимодействие объектов ОА и ОВ.

### Пример 1.26.

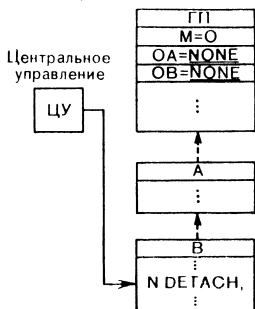
```
begin integer M; ref (A) OA; ref (B) OB;
class A;
  begin
    T: OB:—new B;
    E: detach;
    comment оператор E переводит объект ОА в самостоя-
    тельное состояние и возвращает управление к месту
    генерации, в оператор L;
    ЦИКЛ A: for M:=M+1 while M≤1000 do resume (OB);
    comment увеличили значение M и, если M≤1000, возоб-
    новляем работу объекта ОВ;
    detach;
    comment при M>1000 возвращаем управление в глав-
    ную программу, которая возобновит работу с опера-
    тора K;
  end A;
class B; comment построен аналогично классу A;
  begin N: detach;
    comment оператор N возвращает управление в опера-
    тор T и открепляет объект ОВ от объекта ОА, к кото-
    рому ОВ был прикреплен до исполнения оператора N;
    ЦИКЛ B: for M:=M+1 while M≤1000 do
      resume (OA);
    C: detach;
  end B;
comment операторы L, P, K составляют правила действий
главной программы. Работа симула-программы начинается
с оператора L;
  L: OA:—new A;
comment K моменту исполнения оператора P будут су-
ществовать самостоятельные объекты ОА и ОВ;
  P: resume (OA);
comment Оператор P запускает в работу объект ОА. Пос-
ле этого ГП становится неактивной, ее ЛУ указывает на
оператор K. Объекты ОА и ОВ начинают изменять зна-
чение M и «перебрасывать» управление друг другу;
  K: outtext ('КОНЕЦ РАБОТЫ');
end
```

На рис. 10 изображены структуры данных, соответствующие различным стадиям исполнения программы из примера 1.26.

Программные компоненты, взаимодействующие между собой, подобно объектам ОА и ОВ, принято называть *сопрограммами*. Термин сопрограммы обусловлен их симметричным, равноправным взаимодействием, в отличие от взаимодействия программы и подпрограммы, при котором подпрограмма играет по отношению к программе подчиненную роль.

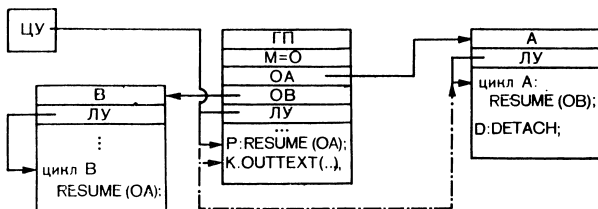
1.6.2. Пример использования сопрограмм. Рассмотрим пример применения средств организации квазипараллельной работы объектов для решения задачи о расстанов-

а) перед исполнением оператора N



Создаваемый объект класса А прикреплён к ГП, а к нему, в свою очередь, прикреплён объект класса В, генерируемый оператором T.OB -NEW B;

б) после исполнения оператора E, но перед исполнением P:



После этого объекты ОА и ОВ передают управление друг другу посредством операторов RESUME(OB) и RESUME(OA) соответственно. При  $M > 1000$  один из операторов DETACH(C или D) в объектах ОА и ОВ передает управление ГП, которая возобновляет работу с оператора N.

Рис. 10. Структура данных для программы из примера 1.26.

ке N ферзей на шахматной доске размером  $N \times N$  так, чтобы они не били друг друга. Решение этой задачи с помощью рекурсивной процедуры, генерирующей допустимые расстановки, приведено в статье [41].

Мы рассмотрим процесс получения допустимых расстановок в виде взаимодействия N ферзей, которые передвигаются по шахматной доске, пытаясь встать так, чтобы не угрожать друг другу. Ферзи при этом представляются объектами, действующими как сопрограммы. Такой подход позволяет, на наш взгляд, получить более наглядную программу, чем в случае использования рекурсивной процедуры.

В данном разделе мы построим декларацию класса QUEEN, объекты которого будут отображать ферзей, передвигающихся по доске в поисках допустимых расстановок. Мы опустим вопросы представления шахматной доски, запоминания и вывода получаемых расстановок, поскольку они не имеют большого значения для описания правил действий ферзей. Полный текст программы с примерами вывода расстановок приводится в Приложении 9. В программе используется представление доски, описанное в статье [11].

Декларация класса QUEEN, приведенная в примере 1.27, отражает следующие правила поведения ферзей на шахматной доске в процессе поиска допустимых расстановок. Каждой вертикали ставится в соответствие свой ферзь. Работа начинается с действий ферзя, связанного с первой вертикалью. Поведение ферзей таково: ферзь, отвечающий вертикали, передвигается по ней и ищет первую свободную клетку, т. е. клетку, которая не бьется ни одним из предыдущих ферзей, расположенных на вертикалях 1, 2, ..., K—1. Если он находит такую клетку на своей вертикали, то он запускает в работу следующего ферзя, который делает то же самое на (K+1)-й вертикали. В том случае, если все клетки K-й вертикали оказались под ударом предыдущих фигур, K-й ферзь возвращается в свое исходное положение и возобновляет работу (K—1)-го ферзя, который должен заново искать свободную клетку, поскольку прежняя расстановка ферзей на вертикалях 1, 2, ..., K—1 оказалась неудовлетворительной, так как K-й ферзь не нашел свободной клетки на своей вертикали.

Поиск очередной расстановки заканчивается, если находит свободную клетку N-й ферзь. В этом случае нужно запомнить полученную расстановку и перейти к поиску следующей расстановки, для чего последний ферзь должен попытаться прежним образом найти еще одну свободную клетку на своей вертикали. Если это не удастся, то он действует точно так же, как и в случае отсутствия свободных клеток, т. е. запускает в работу (N—1)-го ферзя. Очевидно, что все допустимые расстановки будут получены тогда, когда ферзь, расположенный на первой вертикали, дойдет до последней клетки на этой вертикали и, получив сигнал на возобновление работы от второго ферзя, будет вынужден вернуться в исходное положение. Если в этом случае не остановить работу программы, то начнется повторная генерация всех допустимых расстановок.

Каждый ферзь характеризуется двумя целыми числами, указывающими его текущее положение — номером вертикали (атрибут НОМЕР) и номером клетки на этой вертикали (атрибут КЛЕТКА). В исходном положении значение атрибута КЛЕТКА

у всех ферзей равно 1, а атрибут НОМЕР у каждого ферзя указывает на номер вертикали, по которой он будет передвигаться. Будем считать определенными следующие процедуры:

1. **boolean procedure СВОБОДНА** (K, L); **integer** K, L; ...;

Процедура дает значение **true**, если клетке, расположенной на пересечении вертикали K и горизонтали L, не угрожает ни один ферзь, и значение **false** — в противном случае.

2. **procedure ЗАНЯТЬ** (K, L); **integer** K, L; ...;

Процедура ЗАНЯТЬ объявляет занятой клетку с координатами (K, L) и все клетки, лежащие на проходящих через нее вертикали, горизонтали и двух диагоналях. Используется при остановке ферзя на свободной клетке.

3. **procedure ОСВОБОДИТЬ** (K, L); **integer** K, L; ...;

Эта процедура освобождает клетку (K, L), т. е. снимает «угрозу» со всех клеток, доступных из нее за один ход ферзя. Используется при уходе ферзя с занимаемой им клетки.

4. **procedure ЗАПОМНИТЬ РАССТАНОВКУ**; ...;

Выполняет действия по запоминанию полученной расстановки N ферзей и, если это необходимо, обращается к процедуре вывода накопленных расстановок на печать.

Для обращения к объектам, отображающим ферзей, в программе предусмотрен массив ссылок **ref** (QUEEN) **array** ФЕРЗИ [1 : N]. Присваивание значений этому массиву и установка ферзей в начальное положение выполняется оператором цикла, расположенным в главной программе:

```
for I:=1 step 1 until N do  
  ФЕРЗИ[I]:=new QUEEN (I, 1);
```

Вычисление генератора объектов на каждом шаге цикла заканчивается первым оператором **detach** в теле декларации класса QUEEN. По окончании этого цикла все ферзи готовы к работе и следующий оператор главной программы:

```
resume (ФЕРЗИ[1]);
```

запускает в работу первого ферзя, который занимает клетку с координатами (1,1) и «толкает» второго ферзя и т. д. Работа главной программы приостанавливается до тех пор, пока в результате взаимодействия сопрограмм, отображающих поведение ферзей, не будут получены все допустимые расстановки. Главная программа получит управление от первого ферзя в результате выполнения оператора **detach** с меткой ВСЕ, когда этот



ферзь пройдет все клетки своей вертикали. В примере 1.27 приведена декларация класса QUEEN с необходимыми комментариями, а в Приложении 9.1 программа расстановки ферзей и результаты ее работы.

Пример 1.27

```
class QUEEN (НОМЕР, КЛЕТКА); integer НОМЕР, КЛЕТКА;  
begin detach; comment переход в самостоятельное состояние, завершение генерации;  
comment двигаемся по вертикали в поисках свободной клетки;  
ПОИСК: if СВОБОДНА (НОМЕР, КЛЕТКА) then  
begin ЗАНЯТЬ (НОМЕР, КЛЕТКА);  
if НОМЕР < N then resume (ФЕРЗИ[НОМЕР+1]) else  
ЗАПОМНИТЬ РАССТАНОВКУ;  
comment ищем следующую свободную клетку, освобождая занятую;  
ОСВОБОДИТЬ (НОМЕР, КЛЕТКА);  
end;  
СДВИГ: КЛЕТКА:=КЛЕТКА+1; if КЛЕТКА > N then  
begin comment возвращаемся в исходное положение и «толкаем» предыдущего ферзя. Если ферзь — первый, то заканчиваем работу, возвращая управление главной программе (метка ВСЕ);  
КЛЕТКА:=1; if НОМЕР=1 then ВСЕ: detach;  
resume (ФЕРЗИ[НОМЕР-1])  
end;  
go to ПОИСК;  
comment продолжаем поиск свободной клетки;  
end QUEEN;
```

## ГЛАВА 2

# СРЕДСТВА ИМИТАЦИОННОГО МОДЕЛИРОВАНИЯ

В этой главе рассматриваются средства языка симула-67, ориентированные на описание имитационных моделей дискретных систем. Эти средства содержатся в системном классе *simulation*, полное формальное описание которого на языке симула-67 приведено в книге [10]. В данной работе мы сосредоточим внимание на содержательном описании предлагаемых средств моделирования и примерах их применения.

В § 2.1—2.3 рассматривается общая структура класса *simulation*, вводятся понятия процесса, как основного средства отображения объектов моделируемых систем, и оси системного времени, как аппарата синхронизации процессов. Рассматриваются операторы управления процессами, общая организация программы моделирования. § 2.4 и 2.5 посвящены средствам стохастического моделирования и сбора статистических данных о работе модели, а § 2.6 содержит ряд несложных программ моделирования, иллюстрирующих применение рассмотренных средств.

### 2.1. Процессы. Структура класса *simulation*

В системном классе *simulation*, разработанном авторами языка симула-67 [10], определены средства для описания имитационных моделей дискретных систем. Этот класс построен на основе класса *simset* (см. 1.5) и широко использует предложенный там аппарат для работы с упорядоченными множествами, а также средства иерархического описания классов объектов (1.3) и механизм квазипараллельного исполнения объектов (1.6).

Класс *simulation* может служить хорошим примером разработки системы средств, имеющих более конкретную предметную ориентацию (в данном случае — на имитацию дискретных

систем), на базе достаточно общих средств класса *simset*, ориентированных на организацию упорядоченных множеств в любой предметной области. Аналогичным образом, на базе класса *simulation* могут быть разработаны специализированные средства имитационного моделирования, облегчающие создание моделей для некоторой предметной области или определенного класса систем.

В основе средств моделирования, предлагаемых в классе *simulation*, лежат понятия *системного времени и процесса*. Термином системное время в языках моделирования принято обозначать арифметическую величину, принимающую неотрицательные, неубывающие значения, которая в ходе работы модели отображает течение времени в моделируемой системе. Процессом в языке симула-67 называется объект, работа которого привязана к системному времени, т. е. последовательности активных фаз такого объекта соответствует неубывающая последовательность моментов системного времени. Исполнение активной фазы процесса принято называть событием.

События отображают изменения состояния моделируемой системы посредством изменений в состоянии модели. Каждому событию при работе модели соответствует определенный момент системного времени, причем в ходе события системное время остается постоянным. Это означает, что изменения состояния модели происходят мгновенно с точки зрения системного времени. Нескольким событиям в модели может соответствовать один и тот же момент системного времени, что позволяет отображать одновременные события, происходящие в моделируемой системе.

В имитационной модели, написанной на языке симула-67 с использованием средств класса *simulation*, объекты моделируемых систем представляются процессами, а работа отдельных процессов и их взаимодействие отражают функционирование системы. Количество процессов и их взаимосвязи могут меняться в ходе работы модели, что позволяет легко описывать динамические системы с переменной структурой. В роли атрибутов процессов могут выступать переменные различных типов, массивы, ссылки на другие процессы, процедуры. Для описания правил функционирования процессов язык предоставляет алгоритмические возможности, характерные для современных языков программирования высокого уровня, а это дает возможность адекватно отображать объекты со сложными правилами функционирования. В реализациях языка симула-67 на БЭСМ-6 и ЕС ЭВМ [1, 7] предусмотрены также средства интерфейса с другими системами программирования (см. 4.3), позволяющие там, где это удобно, частично или полностью описывать алго-

ритмы поведения процессов на других языках, например, на фортране и пользоваться накопленным на этих языках программным фондом.

Важной особенностью языка симула-67, выгодно отличающей его от других языков моделирования, являются средства структурированного, иерархического описания классов процессов (см. 1.3). Эти средства позволяют для широкого спектра предметных областей описывать типовые модели характерных для них объектов и дают возможность постепенно наращивать, детализировать описания типовых моделей, отслеживая поэтапную конкретизацию представлений о системах в процессе их исследования или проектирования.

В модели параллельно (с точки зрения системного времени) исполняются несколько цепочек событий, каждая из которых соответствует работе отдельного процесса (см. рис. 11). В декларациях классов, задающих атрибуты и правила действий процессов, разработчик модели должен описать их функционирование с локальной точки зрения, т. е. с точки зрения отдельного описываемого процесса. В частности, разработчику достаточно описать чередование событий отдельно для каждого класса процессов, а правильное следование во времени событий, принадлежащих различным цепочкам (синхронизацию процессов), автоматически обеспечивает система моделирования, определенная в классе *simulation*.

Основным инструментом синхронизации процессов, применяемым в классе *simulation*, является так называемый *управляющий список* (УС). Он представляет собой упорядоченный по системному времени набор (см. 1.5.1), членами которого состоят объекты, отображающие запланированные, но еще не исполненные к данному моменту системного времени события. Эти объекты называются уведомлениями о событии. Каждое уведомление содержит информацию о запланированном времени события и ссылку на процесс, очередная активная фаза которого должна исполняться в это время. Первое уведомление в УС указывает на активный в данный момент процесс (текущий процесс), т. е. процесс по операторам правил действий которого идет центральное управление. Время события, указанное в первом уведомлении, равно текущему системному времени (см. рис. 11).

Во время исполнения своей очередной активной фазы процесс может запланировать события для других процессов, а также следующую активную фазу для себя самого. Кроме того, процесс может отменить некоторые из ранее запланированных, но еще не исполненных событий. Планирование и отмена исполнения активных фаз процессов выполняются

специальными операторами, определенными в классе *simulation* (см. 2.2).

При планировании событий происходит включение в УС новых уведомлений или смена позиции уже присутствующих там уведомлений (перепланирование). Отмене события соответствует исключение уведомления из УС. При выполнении

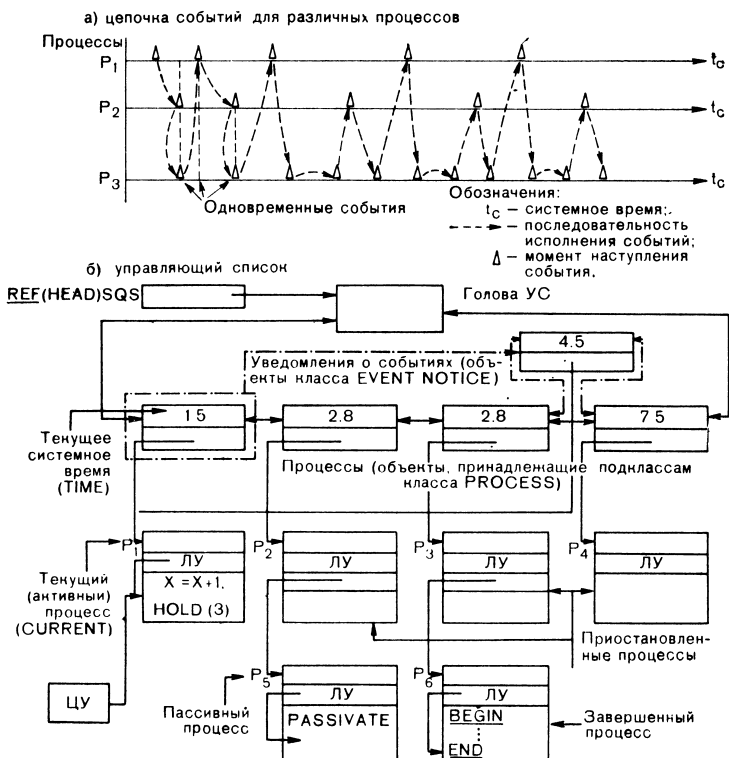


Рис. 11. Структура управляющего списка.

---> Смена позиции уведомления о текущем процессе при исполнении им оператора *hold (3)*.

любых операций над управляющим списком процедуры класса *simulation*, реализующие операторы управления процессами, поддерживают упорядоченность УС по системному времени. Завершив вставку, удаление или перестановку уведомления, эти процедуры запускают в работу тот процесс, на который указывает уведомление, оказавшееся первым в управляющем списке. При этом может произойти изменение текущего си-

стемного времени, если все запланированные на данное время события были выполнены.

В ходе работы модели процессы могут находиться в следующих четырех состояниях: активном, приостановленном, пассивном и завершенном.

Активным в каждый момент времени является только один процесс, на который указывает первое уведомление в УС. Этот процесс в данный момент непосредственно исполняет операторы, описывающие его правила действий. По окончании текущего события активный процесс может перейти в любое из перечисленных состояний.

Для приостановленных процессов характерно наличие ссылающихся на них уведомлений в УС. Это означает, что для каждого из них запланировано событие и оно будет исполнено (т. е. рано или поздно приостановленный процесс станет активным), если, конечно, во время предыдущих событий его никто явно не отменит, или моделирование не будет прекращено.

Пассивные процессы, в отличие от приостановленных, не представлены уведомлениями в УС, т. е. событий для них не запланировано. Однако локальное управление пассивного процесса указывает на один из операторов его правил действий и для него можно запланировать активную фазу, которая начнется с исполнения этого оператора. Иными словами, пассивный процесс может быть переведен в активное или приостановленное состояние. Непосредственно после генерации процесс находится в пассивном состоянии, и его локальное управление указывает на первый исполняемый оператор правил действий.

Завершенный процесс так же, как и пассивный, не представлен в УС, но не может перейти ни в какое другое состояние, т. е. его работа не может быть возобновлена и он присутствует в модели как структура данных. Процесс попадает в заверщенное состояние при выходе управления через завершающий символ `end` его декларации класса или в результате исполнения оператора перехода к нелокальной метке.

Рассмотрим общую структуру класса `simulation`, приведенную на рис. 12. Прежде всего в этом классе определяется набор `SQS (SEQUENCING SET)`, членами которого являются объекты класса `EVENT NOTICE (УВЕДОМЛЕНИЕ О СОБЫТИИ)`, составляющие управляющий список. Создание головы набора `SQS` и занесение туда первого уведомления выполняется операторами тела класса `simulation`, работа которых предшествует началу моделирования.

Декларация класса `EVENT NOTICE` описывает уведомления о событиях, характеризуемые временем события (`ETIME`) и

```

simset class simulation;
begin ref (head) SQS;
  link class EVENT NOTICE (ETIME, PROC);
  real ETIME; ref (process) PROC; ...;
  link class process;
    begin ref (EVENT NOTICE) EVENT;
    boolean TERM
    boolean procedure idle; idle:=EVENT==none;
    boolean procedure terminated; terminated:=TERM;
    real procedure evtime; if  $\neg$  idle then
    evtime:=EVENT.ETIME;
    ref (process) procedure nextev;
      if  $\neg$  idle then nextev:=EVENT.suc qua
      EVENT NOTICE.PROC;
      detach;
      inner;
      TERM:=true; passivate; ERROR;
    end process;
  procedure passivate; ...;
  procedure cancel (X); ref (process) X; ...;
  procedure hold (T); real T; ...;
  procedure wait (S); ref (head) S; ...;
  ref (process) procedure current; current:=SQS.first qua EVENT
  NOTICE.PROC;
  real procedure time; time:=SQS.first qua EVENT NOTICE.ETIME;
  (описания процедур, определяющих семантику операторов
  activate и reactivate)
  comment исполняемые операторы тела класса simulation;
  SQS:=new head; ...
  (другие операторы, выполняющие стандартные начальные
  действия по организации моделирования.)
  inner;
end simulation

```

Рис. 12. Общая структура класса simulation.

ссылкой на процесс (PROC). При планировании очередной активной фазы для пассивного процесса X на системное время T программы класса simulation создают уведомление о событии при помощи генератора объектов

**new EVENT NOTICE (T, X)**

и включают его в УС (набор SQS), пользуясь процедурами класса simset.

Центральное место в классе simulation принадлежит декларации класса process, задающей общие свойства для всех объектов, функционирование которых привязано к системному времени.

Задание конкретных классов процессов выполняется внутри блока с префиксом simulation, где доступны все средства, определенные в этом классе. Для этого достаточно употребить имя process в качестве префикса к декларации определяемого класса. Например, начало программы моделирования работы вычислительного центра, определяющее атрибуты и правила функционирования ЭВМ, может иметь следующий вид.

**Пример 2.1.**

**simulation begin**

**process class ЭВМ (P); real P;**

**comment P — производительность ЭВМ;**

**begin**

*<описание правил действий ЭВМ, привязанное к системному времени.>*

**end ЭВМ;**

**... .**

**end программы**

Отметим, что объекты класса process и его подклассов могут быть членами наборов, поскольку декларация класса process имеет префикс link (см. 1.5), и все объекты этого класса (и его подклассов) будут обладать ссылками для связи с соседями по набору и атрибутами-процедурами для работы с наборами.

Среди атрибутов процессов, заданных в декларации класса process, присутствуют переменные TERM (TERM имеет значение **true** для завершенных процессов) и EVENT (указывает на уведомление, ссылающееся на данный процесс), а также ряд процедур, позволяющих оценить состояние процесса. Например, процедура **idle** имеет значение **true** в том случае, если процесс «бездействует», т. е. находится в пассивном или завершенном состоянии (для такого процесса нет уведомления в УС), а процедура **terminated** дает значение **true** только для завершенных



процессов, т. е. тех, которые выполнили правила действий, заданные пользователем (они представлены в декларации класса `process` символом `inner`), и возобновление их работы вызовет ошибку.

**З а м е ч а н и е.** Атрибуты `TERM`, `EVENT`, `SQS`, как и другие атрибуты-переменные, определенные в классе `simulation`, доступны пользователю только через процедуры, что гарантирует их защиту от некорректных действий программы моделирования.

С помощью процедуры `nextev` процесс может получить ссылку на другой процесс, уведомление о котором находится в УС непосредственно после уведомления для данного процесса. Пользуясь дистанционными обращениями к этой процедуре, можно просмотреть весь управляющий список.

Процедура `evtime` выдает в качестве значения действительное число, равное системному времени, на которое запланировано событие для ее процесса (т. е. для процесса, атрибутом которого является эта процедура).

Оператор `detach` в правилах действий декларации класса `process` возвращает центральное управление к месту генерации процесса, останавливая его локальное управление перед символом `inner`. Фактически это эквивалентно установке локального управления на начало исполняемых операторов правил действий процесса, заданных в теле декларации с префиксом `process`.

Операторы, следующие за `inner`, исполняются после того, как управление проходит через конец декларации класса, имеющей префикс `process`. Они обеспечивают установку признака завершения процесса (`TERM` принимает значение `true`), уничтожение уведомления о данном процессе в УС и запуск в работу очередного процесса, а также выдачу сообщения об ошибке, если будет предпринята попытка возобновить работу завершеного процесса с помощью оператора `resume` (см. 1.6). Использование в модели лишь операторов управления процессами, определенных в классе `simulation` (2.2), исключает возможность появления подобных ошибок. Атрибутами класса `simulation` являются также процедуры `passivate`, `cancel`, `wait`, `hold`, которые позволяют переводить процесс из одного состояния в другое. Для планирования новых событий в классе `simulation` предусмотрены специальные операторы — `activate` и `reactivate`, семантика которых описана с помощью процедуры `activate` и ряда вспомогательных процедур, которые не приведены на рис. 12. Все упомянутые здесь процедуры и операторы `activate` и `reactivate` подробно рассматриваются в § 2.2, посвященном средствам управления процессами.

Значение текущего системного времени можно получить, обратившись к процедуре-функции `time`, а процедура `current` выдает в качестве значения ссылку на текущий процесс.

Операторы, заданные в теле класса `simulation`, обеспечивают выполнение стандартных начальных действий, необходимых для организации работы модели, например, создают набор `SQS`, представляющий управляющий список, заносят в УС первое уведомление со временем события, равным 0, и т. д. После исполнения этих операторов символ `inner` «отправляет» управление на первый исполняемый оператор, заданный пользователем в теле блока с префиксом `simulation`. Этот блок является, по существу, описанием имитационной модели. В § 2.3 подробно рассматриваются вопросы, связанные со структурой программ моделирования, использующих средства класса `simulation`.

## 2.2. Операторы управления процессами

Управление процессами в программах моделирования, использующие средства класса `simulation`, состоит в планировании для них новых активных фаз (событий) и в отмене некоторых из ранее запланированных активных фаз. Планированию активной фазы процесса соответствует создание нового уведомления о событии и внесение его в управляющий список (УС) либо изменение позиции уведомления в УС. При отмене события соответствующее уведомление о событии исключается из УС. Указанные действия над УС задаются с помощью специальных операторов управления процессами, которые определены в классе `simulation`. В 2.2.1 рассматривается часто используемый оператор `hold`, задающий задержку процесса на определенный интервал системного времени, а 2.2.2 и 2.2.3 посвящены операторам, выполняющим планирование и отмену событий.

**2.2.1. Задержка процесса.** При описании работы объектов моделируемых систем, которые отображаются с помощью процессов, часто возникает необходимость задержать дальнейшее исполнение процесса на некоторый интервал системного времени. Посредством таких задержек имитируются, например, затраты времени на выполнение тех или иных действий, предпринимаемых моделируемым объектом, ожидание в течение определенного времени, временные интервалы между изменениями состояния объекта.

Для того чтобы обеспечить исполнение новой активной фазы некоторого процесса через время  $T$  после окончания его текущей активной фазы, достаточно употребить в качестве ее по-

следнего исполняемого оператора оператор

hold (T);

где T — арифметическое выражение, равное требуемому времени задержки. В результате выполнения этого оператора время события в уведомлении о текущем процессе будет увеличено на T, а само оно будет переставлено в УС с первого места в позицию, соответствующую системному времени  $\text{time} + T$ , причем, если в УС уже имеются уведомления с таким же временем события, то переставляемое уведомление помещается после них. При этом текущая активная фаза процесса заканчивается, его локальное управление останавливается перед оператором, следующим за hold, а активным становится процесс, на который ссылается уведомление, ставшее первым в результате вышеуказанной перестановки. Если при выполнении hold (T) оказалось, что в интервале системного времени от time (текущее время) до  $\text{time} + T$  не запланировано ни одного события, то результатом действия этого оператора будет просто изменение текущего системного времени на величину T.

В качестве примера использования оператора hold приведем декларацию класса, которая описывает автомобили, находящиеся в эксплуатации. Каждый из автомобилей проезжает за время DT расстояние DS, затрачивая на это DV литров горючего. Когда кончается горючее, автомобиль заправляется в течение TZ единиц времени, а через каждые 5000 км пробега автомобиль ремонтируется, на что тратится TP единиц времени. Будем считать, что отремонтированный автомобиль заново отсчитывает свой пробег, а после 10 ремонтов его невозможно эксплуатировать более 5000 км, т. е. 11-й ремонт не производится, и автомобиль прекращает активное существование.

Пример 2.2.

```
process class АВТОМОБИЛЬ (VБАКА, DT, DS, DV, TP, TZ,
```

```
real VБАКА, DT, DS, DV, TP, TZ;
```

```
comment VБАКА — объем одной заправки горючим;
```

```
begin real ПРОБЕГ, ГОР; integer КРЕМ;
```

```
comment в процессе работы автомобиля значение атрибута ГОР равно текущему количеству горючего, а значение КРЕМ — количеству проведенных ремонтов;
```

```
ГОР:=VБАКА; РАБОТА: ПРОБЕГ:=ПРОБЕГ+DS;
```

```
ГОР:=ГОР-DV;
```

```
hold (DT);
```

```
if ПРОБЕГ $\geq$ 5000 then goto РЕМОНТ;
```

```
if ГОР=0 then goto ЗАПРАВКА else goto РАБОТА;
```

РЕМОНТ: if КРЕМ=10 then goto СПИСАТЬ;  
 hold (TP); КРЕМ:=КРЕМ+1; ПРОБЕГ:=0;  
 goto РАБОТА; ЗАПРАВКА: hold (TZ);  
 ГОР:=ВБАКА; goto РАБОТА; СПИСАТЬ:  
 end АВТОМОБИЛЬ;

2.2.2. Планирование новых событий. Планирование новых событий (активных фаз процессов) в программах моделирования, написанных на языке симула-67, выполняется с помощью так называемых планирующих операторов, определенных в классе simulation. Как уже указывалось, при планировании события производится включение в управляющий список нового уведомления или смена позиции одного из уже присутствующих уведомлений.

Для того чтобы запланировать событие, нужно задать два атрибута уведомления о событии: момент системного времени, в который оно должно произойти, и процесс, очередная активная фаза которого будет составлять содержание этого события. Эти два атрибута, заданные явным или неявным образом, служат аргументами планирующих операторов. Их синтаксис имеет следующий вид:

$$\left\{ \begin{array}{l} \text{activate} \\ \text{reactivate} \end{array} \right\} X \left\{ \begin{array}{l} \langle \text{НУСТО} \rangle \\ \left\{ \begin{array}{l} \text{at } T \\ \text{delay } T \\ \text{after } Y \\ \text{before } Y \end{array} \right\} [\text{prior}] \end{array} \right\}$$

где X — объектное выражение, обозначающее процесс, для которого планируется событие; T — арифметическое выражение, значение которого трактуется в случае at T как абсолютное системное время события, а в случае delay T — как время задержки планируемого события относительно текущего момента системного времени (delay T эквивалентно at time+T); Y — объектное выражение, обозначающее активный или приостановленный процесс.

Рассмотрим более подробно семантику различных форм планирующих операторов, начинающихся с ключевого слова activate. Эти операторы выполняют планирование событий только для пассивных процессов, создавая для них новые уведомления в УС, в отличие от операторов reactivate, которые воздействуют и на процессы, находящиеся в активном или приостановленном состоянии. Таким образом, действия, выполняемые операторами activate и reactivate при планировании событий для пассивных процессов полностью совпадают, а работа оператора reactivate над активным или приостановленным процессом отличается

лишь тем, что уже имеющееся в УС уведомление о событии для данного процесса предварительно исключается из УС, т. е. этот процесс переводится в пассивное состояние.

**Оператор вида**

**activate X at T;**

в котором явно указаны системное время планируемого события (T) и процесс, подлежащий планированию (X), выполняет следующие действия:

а) создает уведомление о событии, ссылающееся на процесс X и имеющееся время события, равное значению арифметического выражения T;

б) вставляет созданное уведомление в управляющий список перед всеми уведомлениями с временами события большими, чем T и после всех уведомлений с временами события меньшими или равными T.

Если планирующий оператор снабжен указанием о планировании с приоритетом, т. е. имеет вид

**activate X at T prior;**

то уведомление о событии для процесса X вставляется перед всеми уведомлениями с временами события, большими или равными T и после всех уведомлений с временами события меньшими, чем T. Это означает, что при планировании события с приоритетом оно будет исполнено перед другими событиями, ранее запланированными на этот же момент системного времени. При планировании без приоритета одновременные события исполняются в том порядке, в котором соответствующие уведомления попадали в управляющий список. Работа планирующих операторов с явным указанием времени события проиллюстрирована на рис. 13.

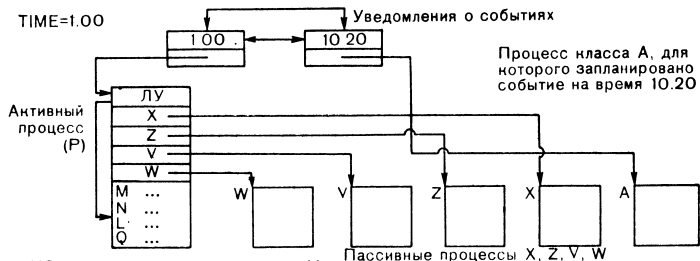
При планировании событий можно задать не время, а позицию нового уведомления в УС. Для этого используются операторы вида

**activate X    { before  
                  after } Y;**

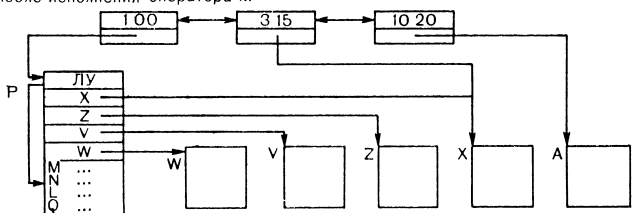
При работе такого оператора создается уведомление о событии, ссылающееся на процесс X, которое вставляется в УС непосредственно перед (в случае **before**) или непосредственно после (в случае **after**) уведомления о событии для процесса Y. В обоих случаях время события для процесса X полагается равным времени события для процесса Y (см. рис. 14). Если Y не представлен в УС, то операторы **activate** и **reactivate** с пассивным процессом X никаких действий не производят, а

а) исходное состояние управляющего списка (УС) перед исполнением активным процессом Р следующих операторов:

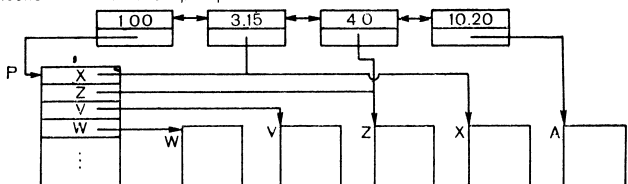
M ACTIVATE X AT 3.15; N ACTIVATE Z AT 4.00;  
L ACTIVATE V DELAY 3.15, Q: ACTIVATE W DELAY 3.15 PRIOR



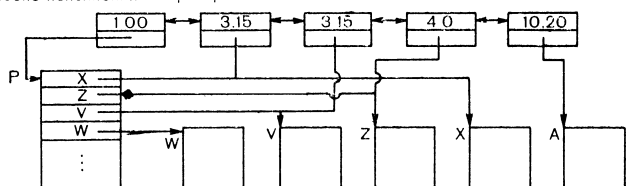
б) УС после исполнения оператора М:



в) УС после исполнения оператора N:



г) УС после исполнения оператора L:



д) УС после исполнения оператора Q:

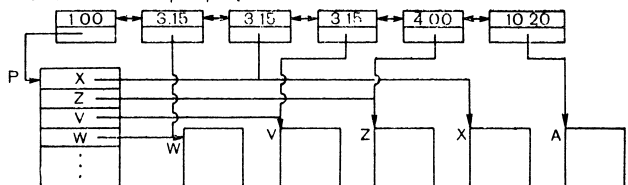
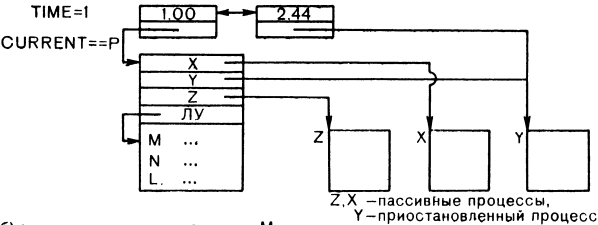


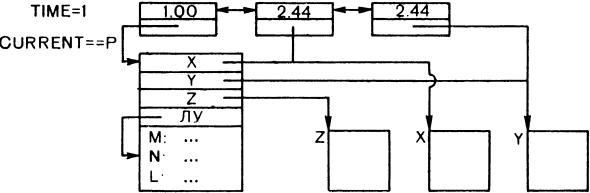
Рис. 13. Планирование с явным указанием времени события.

а) исходное состояние УС перед исполнением активным процессом Р следующих операторов

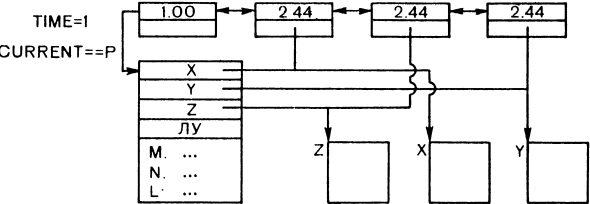
M ACTIVATE X BEFORE Y;  
 N ACTIVATE Z AFTER X;  
 L: REACTIVATE X BEFORE CURRENT;



б) после исполнения оператора M



в) после исполнения оператора N



г) УС после исполнения оператора L (непосредственное планирование, оператор L эквивалентен REACTIVATE X)

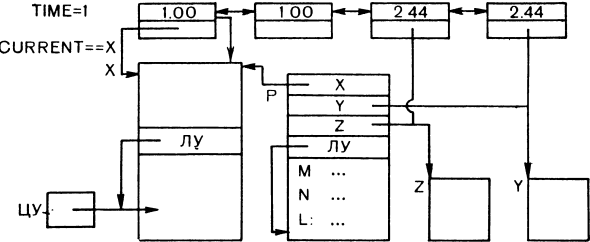


Рис. 14. Планирование с явным указанием позиции уведомления о событии в УС.

оператор **reactivate**  $X \left\{ \begin{array}{l} \text{before} \\ \text{after} \end{array} \right\} Y$ ; где  $X$  — активный или приостановленный процесс, исключит уведомление об  $X$  из УС, т. е. сделает процесс  $X$  пассивным.

Рассмотрим ситуацию, когда новое уведомление, вставляемое в УС, оказывается первым. Это происходит, например, при исполнении планирующих операторов вида

**activate**  $X$  at time prior;

где  $X$  — пассивный процесс, или операторов

**reactivate**  $X$  before current;

где  $X$  — любой процесс, кроме заверщенного. В этих случаях говорят, что планирующий оператор задает непосредственное планирование для процесса  $X$  (см. рис. 14, а, г).

Непосредственное планирование можно задать и с помощью операторов, имеющих форму

**activate**  $X$ ; и **reactivate**  $X$ ;

представляющих собой краткую запись вышеприведенных планирующих операторов.

Обозначим через  $P$  процесс, исполняющий оператор непосредственного планирования для процесса  $X$ . В результате выполнения этого оператора процесс  $X$  становится активным, а процесс  $P$  — приостановленным, причем его локальное управление указывает на оператор, следующий за оператором непосредственного планирования, а уведомление о процессе  $P$  оказывается вторым в УС. Когда окончится активная фаза процесса  $X$ , процесс  $P$  возобновит свою работу, если, конечно, процесс  $X$  не сделает его пассивным или не перепланирует его на другой момент времени. В случае, если при непосредственном планировании процессы  $X$  и  $P$  совпадают, никаких действий такой оператор не производит. Рисунок 14 иллюстрирует действие некоторых только что рассмотренных операторов.

Операторы управления процессами, рассмотренные в данном разделе, позволяют запрограммировать любой алгоритм планирования событий. В связи с этим оператор **hold** ( $T$ ), задающий задержку выполнения процесса на время  $T$  (см. 2.2.1), не является необходимым и служит лишь краткой формой записи оператора **reactivate** current delay  $T$ ;

В качестве примера использования планирующих операторов приведем декларацию класса, описывающую процесс, который каждые  $T$  единиц времени генерирует по одному объекту класса АВТОМОБИЛЬ (см. пример 2.2) и обеспечивает начало движения каждого автомобиля через 10 единиц времени после



его создания. Будем считать, что все автомобили обладают одинаковыми начальными значениями атрибутов ВБАКА, DT, DS, DV, TP, TZ.

### Пример 2.3.

```
process class ГЕНЕРАТОР АВТОМОБИЛЕЙ (T); real T;  
begin  
  РАБОТА: activate new АВТОМОБИЛЬ (40, 1, 60, 6.0,  
    400, 0.5) delay 10;  
  hold (T); goto РАБОТА;  
end;
```

Фрагмент программы, приведенный в примере 2.4, обеспечивает создание процесса, с именем ГЕН, генерирующего автомобили каждые 40 единиц времени и начинающего работу в момент времени 365.

### Пример 2.4.

```
ref (ГЕНЕРАТОР АВТОМОБИЛЕЙ) ГЕН;  
ГЕН:—new ГЕНЕРАТОР АВТОМОБИЛЕЙ (40);  
activate ГЕН at 365;  
. . . . .
```

**2.2.3. Отмена событий.** В процессе работы имитационных моделей часто бывает необходимо отменять некоторые из ранее запланированных событий, если в ходе моделирования сложились условия, делающие невозможными их осуществление в намеченный момент системного времени. Так, например, в модели предприятия следует предусмотреть отмену активной фазы процесса, описывающего сборку изделия, если к определенному моменту времени к месту сборки не поступили необходимые узлы.

Для отмены запланированных активных фаз процессов в классе simulation предусмотрен оператор

**cancel (X);**

где X — объектное выражение, обозначающее процесс, чья активная фаза должна быть отменена. В результате работы этого оператора из УС исключается уведомление о событии, ссылающееся на процесс X, т. е. этот процесс становится пассивным. Если процесс X уже пассивен или завершен, то оператор не производит никаких действий. При исключении из УС уведомления о приостановленном процессе его локальное управление не изменяется, а если аргумент оператора cancel указывает на текущий процесс, то его локальное управление устанавливается на оператор, следующий за оператором cancel. При этом активным становится тот процесс, уведомление о котором стало первым после исключения уведомления о текущем процессе.

Для перевода текущего процесса в пассивное состояние можно пользоваться также оператором

**passivate;**

который эквивалентен оператору **cancel (current)** и часто используется в моделях для отображения прекращения работы какого-либо объекта на неопределенное время.

Если перед тем, как остановить работу некоторого процесса, его нужно внести в набор, отображающий, например, какую-нибудь очередь, то выполнить эти действия можно с помощью одного оператора

**wait (S);**

где **S** — объектное выражение, обозначающее объект класса **head** (голову набора). В результате выполнения этого оператора текущий процесс будет включен в конец набора **S** и переведен в пассивное состояние. Оператор **wait (S)** эквивалентен составному оператору

**begin current. into (S); passivate end;**

Рассмотрим в качестве примера использования средств отмены событий описание процедуры **ОТМЕНИТЬ (T1, T2)**, которая отменяет все события, запланированные в интервале системного времени от **T1** до **T2**, за исключением текущего события.

**Пример 2.5**

```
procedure ОТМЕНИТЬ (T1, T2); real T1, T2;  
  begin ref (process) X, Y;  
    for X:—current, X while X.evtime < T1 do X:—X.nextev;  
    Y:—X;  
    for X:—Y while X.evtime ≤ T2 do  
      begin  
        Y:—X.nextev; cancel (X);  
        if Y == none then goto КОНЕЦ;  
      end;  
    КОНЕЦ;  
  end ОТМЕНИТЬ;
```

Первый оператор цикла в теле процедуры **ОТМЕНИТЬ** устанавливает переменную **X** на первое уведомление с временем события большим или равным **T1**. Второй цикл исключает из УС все уведомления с временами события меньшими или равным, чем **T2**. Работа этого цикла заканчивается, если будет встречено уведомление с временем события большим, чем **T2**, либо достигнут конец управляющего списка. Вспомогательная переменная **Y** устанавливается в теле цикла на процесс, уве-

домление о котором следует за уведомлением, исключаемым из УС оператором cancel (X).

В примере 2.6 представлены декларации классов ПОКУПАТЕЛЬ и КАССИР описывающие следующие правила поведения покупателя и кассира в универсаме: покупатель за время ТПОК делает КП покупок и оплачивает их у кассира за время, пропорциональное КП, Время оплаты одной покупки — С. Покупатель фиксирует в атрибуте ТПЛ время, затраченное на ожидание в очереди и оплату покупок, а кассир подсчитывает суммарное время простоя из-за отсутствия покупателей (атрибут ПРОСТОЙ).

#### Пример 2.6.

```
ref (head) ОЧЕРЕДЬ; ref (КАССИР) КАССА;  
process class ПОКУПАТЕЛЬ (ТПОК, КП); real ТПОК;  
integer КП;  
  begin real ТВХ, ТПЛ;  
    hold (ТПОК);  
    into (ОЧЕРЕДЬ); activate КАССА delay 0;  
    ТВХ:=time; passivate;  
    comment покупатель встал в очередь и перешел в пассивное состояние, отметив время входа в очередь, и «толкнув» кассира;  
    ВЫХОД: ТПЛ:=time—ТВХ;  
    comment Покупатель расплатился, вычислил время, затраченное на ожидание в очереди и оплату покупок;  
  end ПОКУПАТЕЛЬ;  
process class КАССИР (С); real С;  
  comment С — время, затрачиваемое кассиром на оформление одной покупки;  
  begin real ТНАЧПР, ПРОСТОЙ; ref (ПОКУПАТЕЛЬ) П;  
  РАБОТА: for П:— ОЧЕРЕДЬ. first while П!=none do  
    begin  
      hold (П. КП*С); П. out; activate П;  
      comment кассир обслужил покупателя, удалил его из очереди и активировал покупателя, дав ему возможность продолжать свои действия;  
    end обслуживания покупателей;  
    ОТДЫХ: ТНАЧПР:=time; passivate;  
    КОНЕЦ ОТДЫХА: ПРОСТОЙ:=ПРОСТОЙ+(time—ТНАЧПР);  
    goto РАБОТА;  
  end КАССИР;
```

В примере 2.6 предполагается, что процесс, отображающий кассира, имеет имя КАССА, а на набор, соответствующий очереди, ссылается переменная ОЧЕРЕДЬ.

## Оператор

**activate КАССА delay 0;**

в теле декларации класса ПОКУПАТЕЛЬ планирует возобновление работы кассира, если в момент подхода покупателя к кассе кассир простаивал из-за отсутствия покупателей. Работа процесса класса КАССИР начинается в этом случае с оператора с меткой КОНЕЦ ОТДЫХА, а значение ТПД у подошедшего покупателя окажется равным С\*КП. Указанный оператор **activate** не окажет никакого воздействия на процесс КАССА, если он находится в приостановленном состоянии, т. е. выполняет оператор **hold** (П. КП\*С), имитирующий обслуживание покупателя.

### 2.3. Структура программы моделирования

В данном разделе мы рассмотрим вопросы общей организации программы моделирования, инициализации работы модели, а также некоторые варианты задания условий проведения имитационных экспериментов.

Описание модели на языке симула-67 оформляется в виде блока с префиксом **simulation**. Типичная структура программы моделирования приведена на рис. 15. Описания блока **simulation begin...end** определяют декларации классов для процессов, которые будут функционировать в модели, задают переменные, массивы, процедуры, являющиеся общими для всех процессов. Операторы тела блока с префиксом **simulation** обычно используются для инициализации работы модели и выдачи результатов моделирования.

Работа блока с префиксом **simulation** начинается с операторов, заданных в теле класса **simulation** (см. 2.2.1), которые выполняют стандартные действия, необходимые при инициализации работы любой модели, например, создание управляющего списка и занесение туда уведомления с нулевым временем события, ссылающегося на процесс, называемый Главная программа. С точки зрения пользователя атрибутами этого процесса являются все переменные, массивы, процедуры, классы, описанные в блоке с префиксом **simulation**, а правила действий определяются операторами этого блока. Процесс Главная программа имеет две особенности:

- 1) его первая активная фаза выполняется в нулевой момент системного времени, причем ее планирование обеспечивается не пользователем, а операторами системного класса;

- 2) атрибуты главной программы непосредственно доступны из всех процессов и других объектов, определенных в блоке с префиксом **simulation**.

В остальном главная программа ведет себя точно так же, как и другие процессы модели, в частности, исполнение ее правил действий может быть задержано с помощью оператора **hold**, главную программу можно перевести в пассивное состояние, а затем запланировать возобновление ее работы операторами **activate** или **reactivate** и т. д.

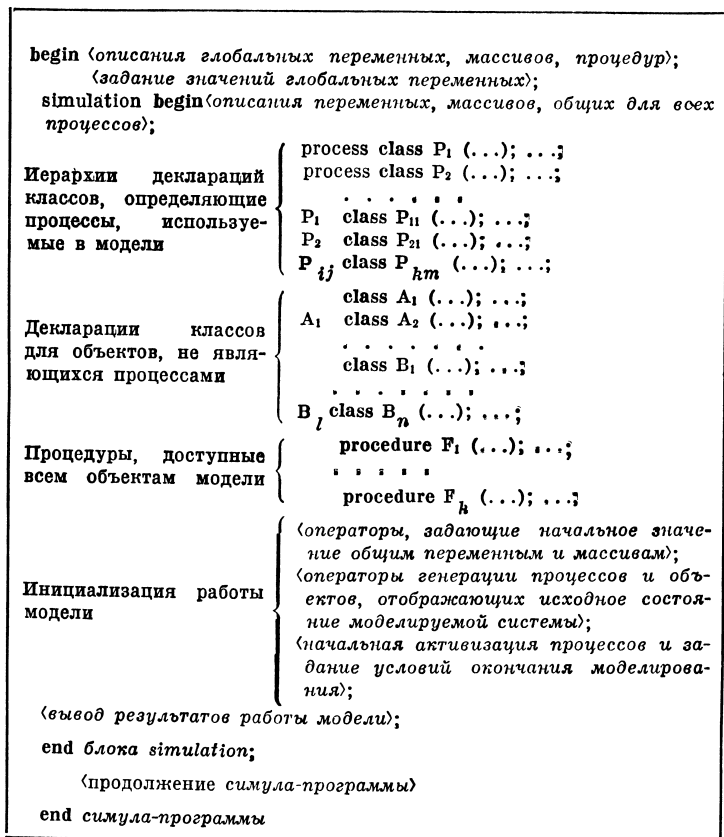


Рис. 15. Типичная структура программы моделирования на языке симула-67.

Для разработчика модели инициализация моделирования состоит в задании начальных значений общим переменным и массивам, генерации процессов и других объектов, отображающих исходное состояние моделируемой системы, запуск этих процессов в работу и указании условий окончания работы

модели. Указанные действия могут быть описаны при помощи всех алгоритмических средств языка симула-67.

Работа модели заканчивается при выходе управления через символ `end`, замыкающий блок с префиксом `simulation`, или при передаче управления на метку, расположенную вне этого блока. При этом уничтожается управляющий список и все объекты, описанные в блоке с префиксом `simulation`. В связи с этим операторы, выполняющие выдачу результатов моделирования, удобно располагать в конце этого блока, когда еще можно пользоваться определенными в нем переменными, массивами, процедурами, атрибутами доступных объектов.

Операторы симула-программы, выполняемые после окончания моделирования, могут повторно передавать управление блоку с префиксом `simulation`, обеспечивая тем самым многократный прогон модели для различных значений глобальных переменных. Иными словами, программа моделирования может быть составной частью симула-программы и взаимодействовать с ней через глобальные переменные, массивы, процедуры, наборы данных и т. д., определяющие условия проведения имитационных экспериментов с моделью. В программе может присутствовать несколько блоков с префиксом `simulation`, описывающих разные модели, которые могут взаимно использовать результаты работы друг друга.

Рассмотрим конкретный пример, иллюстрирующий структуру имитационной модели на языке симула-67, и продемонстрируем на этом примере некоторые варианты организации экспериментов с моделью.

В примере 2.6 (см. 2.2.3) приведены декларации классов ПОКУПАТЕЛЬ и КАССИР, описывающие поведение покупателей и правила работы кассиров в магазине самообслуживания. Для того чтобы получить законченную симула-программу, исполнение которой обеспечит проведение одного или нескольких имитационных экспериментов над моделью магазина, эти декларации классов должны быть охвачены описаниями и операторами, задающими исходные данные для работы модели, условия окончания моделирования и обеспечивающими выдачу его результатов.

Пусть требуется промоделировать работу магазина самообслуживания с одной кассой, нагруженного потоком покупателей, входящих в магазин через случайные промежутки времени, равномерно распределенные в интервале от 3 до 7 минут. Время, затрачиваемое покупателем на покупки, и количество покупок будем также считать случайными величинами, распределенными равномерно в интервалах  $[10, 20]$  и  $[4, 12]$  соответственно. Положим, что кассир тратит 30 секунд на расчет

за одну покупку, и промоделировать нужно 8 часов работы магазина.

Для организации моделирования нам понадобится, в дополнение к покупателям и кассиру, еще один процесс, в задачу которого будет входить генерация и запуск в работу процессов класса ПОКУПАТЕЛЬ через случайные промежутки времени, распределенные в соответствии с заданным законом. Работа этого процесса может быть описана с помощью следующей декларации класса.

Пример 2.7.

**process class ГЕНЕРАТОР;**

**begin**

**СОЗДАНИЕ:** **activate new** ПОКУПАТЕЛЬ (**uniform** (10, 20, U1), **randint** (4, 12, U2));

**comment** генератор создал и активизировал нового покупателя со случайными значениями атрибутов;

**hold** (**uniform** (3, 7, U3));

**comment** задержка на случайное время, равномерно распределенное в интервале от 3 до 7;

**goto** СОЗДАНИЕ;

**comment** переход на генерацию следующего покупателя;

**end ГЕНЕРАТОР;**

В декларации класса ГЕНЕРАТОР использованы определенные в языке симула-67 процедуры получения случайных чисел, распределенных по равномерному закону: **uniform** (A, B, U) дает вещественные числа в интервале [A, B], а результатом процедуры **randint** (M, N, U) служат целые числа, равномерно распределенные в интервале [M, N]. Фактические параметры U1, U2, U3 (целые переменные) обеспечивают взаимную независимость трех потоков псевдослучайных чисел, если их начальные значения различны и являются целыми, положительными нечетными числами. Более подробно встроенные процедуры случайного выбора рассматриваются в § 2.4.

В примере 2.8 приведена законченная симула-программа, обеспечивающая моделирование работы магазина в течение 8 часов и выдающая в качестве результата суммарное время простоя кассира. Предполагается, что единица системного времени соответствует одной минуте работы магазина. Тела деклараций классов те же, что и в примерах 2.6, 2.7.

Пример 2.8.

1. **begin real** ТМОД; **integer** U1, U2, U3;
2. ТМОД:=480;
3. U1:=1; U2:=3; U3:=5;

4. **simulation begin ref (КАССИР) КАССА; ref (head)**  
**ОЧЕРЕДЬ;**
5. **process class КАССИР (C); ...;**
6. **process class ПОКУПАТЕЛЬ (ТПОК, КП); ...;**
7. **process class ГЕНЕРАТОР; ...;**
8. **ОЧЕРЕДЬ:—new head;**
9. **КАССА:—new КАССИР (0.5);**
10. **activate new ГЕНЕРАТОР delay 0; activate КАССА delay 0;**
11. **hold (ТМОД);**
12. **outtext ('ПРОСТОЙ КАССИРА='); outfix (КАССА.**  
**ПРОСТОЙ, 2, 10);**
13. **end модели;**
14. **end программы**

Рассмотрим назначение каждой строки приведенной программы.

Строка 1. Начало самого внешнего блока программы. Описываются глобальные переменные ТМОД, U1, U2, U3, определяющие условия работы модели.

Строка 2. Задание конкретного значения переменной ТМОД, определяющей интервал времени, в течение которого моделируется работа магазина.

Строка 3. Задание начальных значений переменным, используемым для получения псевдослучайных чисел.

Строка 4. Начало блока с префиксом **simulation**, содержащего описание модели. Описываются ссылочные переменные КАССА и ОЧЕРЕДЬ для обозначения процесса, отображающего кассира, и набора, соответствующего очереди покупателей.

Строки 5—7. Декларации классов КАССИР, ПОКУПАТЕЛЬ, ГЕНЕРАТОР (см. примеры 2.6, 2.7).

Строка 8. Создание головы набора ОЧЕРЕДЬ.

Строка 9. Создание процесса класса КАССИР с временем расчета за одну покупку равным 0.5 мин (атрибут C в декларации класса КАССИР). Присваивание созданному процессу имени КАССА.

Строка 10. Создание генератора покупателей и планирование его первой активной фазы на нулевой момент системного времени вслед за текущей активной фазой главной программы, планирование начала работы кассира.

Данная строка завершает подготовку модели к работе: созданы процессы, отображающие исходное состояние моделируемой системы, и запланировано начало их работы.

Строка 11. Задержка дальнейшего исполнения главной программы на время ТМОД. Центральное управление переходит на процесс класса ГЕНЕРАТОР, а локальное управление главной



программы устанавливается на строку 12. Теперь работа программы вплоть до момента времени ТМОД будет состоять во взаимодействии процессов, отображающих генератор, покупателей, приходящих в магазин, и кассира, выполняющего расчет с покупателями.

Строка 12. Вывод результатов моделирования. В результате исполнения приведенных операторов будет напечатана следующая строка (в предположении, что кассир простаивал 45 мин 30 с).

ПРОСТОЙ КАССИРА=45.50

Операторы ввода-вывода, имеющиеся в языке симула-67, подробно рассматриваются в главе 3.

Строка 13. Конец блока с префиксом simulation. При выходе управления через этот символ end моделирование прекращается независимо от того, есть ли в управляющем списке уведомления о запланированных, но еще не исполненных событиях.

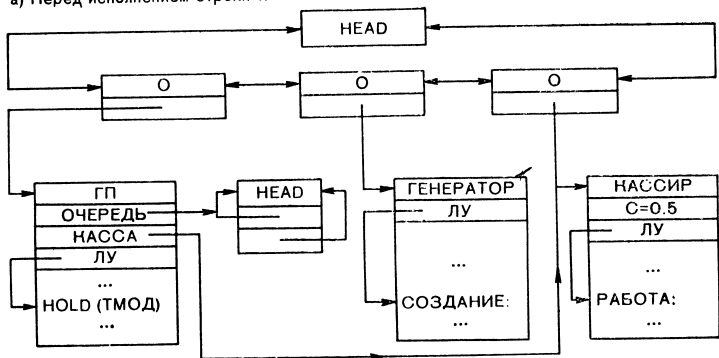
Строка 14. Конец симула-программы, возврат управления в операционную систему.

На рис. 16 схематически изображены состояния управляющего списка при исполнении нескольких активных фаз в начале работы модели. Предполагается, что в результате генерации случайных чисел определились следующие времена появления покупателей и значения их атрибутов: первый покупатель входит в магазин в нулевой момент времени и делает 5 покупок за 10 мин, второй покупатель появляется через 6 мин и делает 8 покупок за 11 мин, третий покупатель приходит через 7 мин после прихода второго.

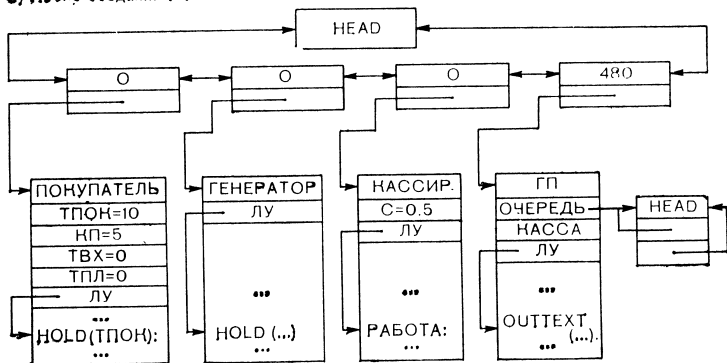
В примере 2.8 мы рассмотрели самый простой случай организации имитационного эксперимента: однократный прогон модели с определенными значениями параметров модели и с заданным временем моделирования. Богатые алгоритмические возможности языка симула-67 позволяют задавать любые условия проведения имитационных экспериментов. Ниже рассматриваются некоторые варианты организации многократных прогонов модели магазина самообслуживания.

Пусть требуется исследовать работу магазина, например, определить время простоя кассира, при различных значениях времени, затрачиваемого кассиром на оформление одной покупки, сделанной покупателем. Эксперименты с моделью удобно проводить, если исходные данные задаются не в тексте программы, а вводятся в процессе ее работы, что позволяет не проводить каждый раз трансляцию программы (при условии, что она записана в архив вычислительной системы). Допустим, что нужно промоделировать 8 часов работы магазина при

а) Перед исполнением строки 11:



б) После создания и активизации первого покупателя:



в) Перед созданием второго покупателя (первый покупатель исполняет оператор HOLD(ТПОН), имитирующий докупку товаров):

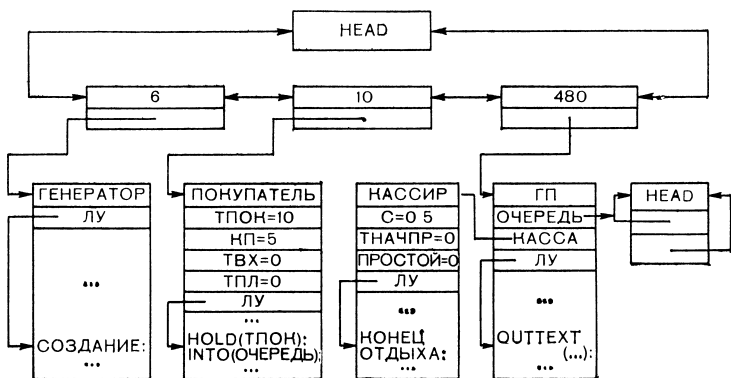
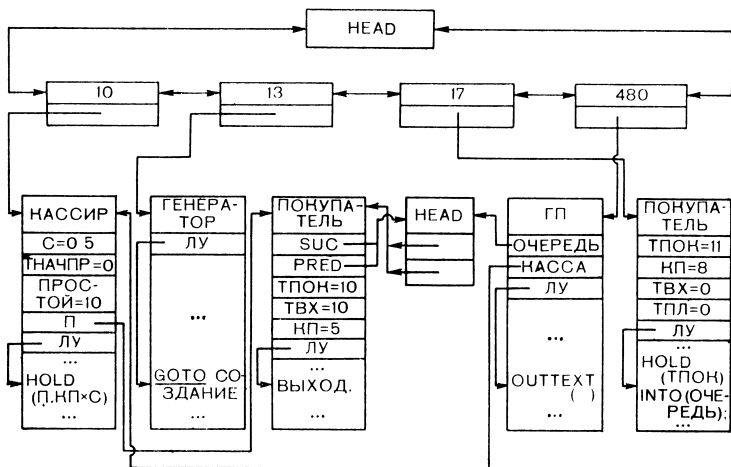


Рис. 16. а — в. Состояния управляющего списка при работе модели магазина,

г) Первый покупатель закончил покупки и активизировал кассира, который начал его обслуживать:



д) Кассир закончил расчет с первым покупателем, который покидает магазин:

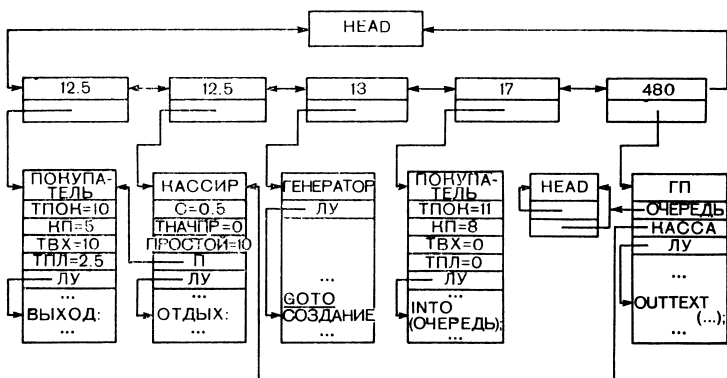


Рис. 16. г, д. Состояния управляющего списка при работе модели магазина.

значениях скорости работы кассира от  $CMIN$  мин/покупку до  $CMAH$  с шагом изменения параметра  $DC$ . Для этого в программу примера 2.8 достаточно внести следующие изменения:

— дополнить описания самого внешнего блока программы, добавив после строки 1 строку 1.1:

1.1. real  $CMIN$ ,  $CMAH$ ,  $DC$ ,  $CT$ ;

— ввести значения параметров CMIN, CMAX, DC:

1.2. CMIN:=inreal; CMAX:=inreal; DC:=inreal;

— внести блок с префиксом simulation в тело цикла:

3.1. for CT:=CMIN step DC until CMAX do

3.2. begin outtext ('ВРЕМЯ ОПЛАТЫ ПОКУПКИ=');

outfix (CT, 2, 6);

simulation begin ... end модели;

13.1. end цикла;

— изменить фактический параметр при генерации процесса класса КАССИР, т. е. строку 9 в примере 2.8 заменить на

9. КАСКА:—new КАССИР (CT);

При работе программы из примера 2.8 с указанными изменениями при каждом исполнении тела цикла будет производиться печать очередного значения времени оплаты одной покупки, а затем будет выполняться моделирование 8 часов работы магазина, которое каждый раз завершается печатью полученного времени простоя.

В качестве условия окончания моделирования можно задавать не только истечение некоторого интервала системного времени, но и достижение заданных ситуаций или определенных условий при работе модели. Пусть, например, моделирование работы магазина (пример 2.8) требуется проводить до тех пор, пока время простоя кассира не превысит 60 минут, но не более, чем в течение времени ТМОД. Для этого достаточно предусмотреть активизацию главной программы в момент достижения заданного условия. Чтобы сослаться на процесс Главная программа в блоке с префиксом simulation нужно описать ссылочную переменную, например ГП, добавив ее описание после строки 4 в примере 2.8:

4.1. ref (process) ГП;

и присвоить ей значение во время первой активной фазы главной программы:

8.1. ГП:—current;

При этом в правилах действия кассира (пример 2.6) нужно предусмотреть активизацию процесса ГП, если время простоя превысило 60 мин. Для этого после оператора с меткой КОНЕЦ ОТДЫХА следует вставить оператор if ПРОСТОЙ > 60 then re-activate ГП, который немедленно запустит в работу главную программу, начиная со строки 12 (пример 2.8), обеспечивающей

выдачу результатов моделирования. Чтобы узнать момент системного времени, в который было прекращено моделирование, после строки 12 можно поместить оператор вывода

12.1. outfix (time, 2, 10);

который напечатает интересующее значение.

Если за время ТМОД простой кассира не превысит 60 мин, то процесс ГП автоматически станет активным через ТМОД единиц системного времени после начала моделирования, поскольку его первая активная фаза закончилась исполнением оператора hold (ТМОД) (строка 11 в примере 2.8).

Одна симула-программа может содержать несколько моделей, взаимодействующих друг с другом через глобальные переменные и наборы данных. В примере 2.9 приведена возможная структура симула-программы, в которой результаты работы одной модели используются как исходные данные для другой.

Пример 2.9.

**begin real P, C;**

*<вычисление значения параметра P>;*

**simulation begin**

*<описание модели 1, зависящей от параметра P. Результатом моделирования служит значение параметра C>;*

**end модели 1;**

**simulation begin**

*<описание модели 2, зависящей от параметра C>;*

**end модели 2;**

**end программы**

Достаточно мощные алгоритмические средства языка симула-67 позволяют непосредственно использовать имитационные модели при решении задач оптимизации сложных систем в тех случаях, когда вычисление целевой функции, оценивающей качество системы, может быть наиболее эффективно проведено в процессе имитации ее работы в течение некоторого промежутка времени. Один из вариантов структуры симула-программы, выполняющей поиск оптимального значения параметра P для системы S, при котором достигается экстремум целевой функции  $F(P)$ , приведен в примере 2.10.

Пример 2.10.

**begin real P;**

*<вычисление начального значения параметра P>;*

**МОДЕЛЬ: simulation begin**

*<описание имитационной модели системы S. В ходе моделирования определяется значение целевой функции  $F(P)$  для текущего значения P>;*

```

end модели S;
<оценка полученного значения F(P)>;
if <экстремум F(P) не достигнут> then
  begin <вычисление очередного приближения для P>;
  goto МОДЕЛЬ
  end;
<вывод оптимального значения P>
end программы оптимизации

```

## 2.4. Средства стохастического моделирования

Для задания различного рода случайных факторов при имитации работы систем в языке симула-67 определен набор встроенных процедур случайного выбора, позволяющих задавать получение псевдослучайных чисел, распределенных по любому нужному закону.

В основе метода получения псевдослучайных чисел, применяемого в языке симула-67, лежит известное положение теории вероятностей о том, что случайные числа  $X$  с любым заданным законом распределения  $F(X)$  можно получить с помощью функционального преобразования случайных чисел  $Y$ , распределенных равномерно на интервале  $[0, 1]$ . Для этого достаточно положить  $X = F^{-1}(Y)$ .

Для получения последовательностей псевдослучайных чисел, равномерно распределенных на интервале  $[0, 1]$ , в языке применяется следующая процедура, называемая процедурой главной выборки:

```

real procedure psrand (U); name U; integer U;
begin integer R;
  R:=U*5** (2*P+1);
  U:=R-(R÷2**N)*2**N;
  psrand:=U/2**N;
end;

```

где  $P$  и  $N$  — целые константы, значения которых зависят от максимального целого числа, представимого на той ЭВМ, где исполняется симула-программа. При многократных обращениях к процедуре `psrand` будет выдаваться последовательность чисел  $A_1, A_2, \dots, A_i, \dots$ , которая однозначно определяется начальным значением ( $U_0$ ) переменной, служащей фактическим параметром обращения к процедуре главной выборки.

В теории вероятностей показывается, что если  $U_0$  — положительное целое нечетное число, то такими же будут и все следующие значения этой переменной  $U_1, U_2, \dots, U_i, \dots$ , полу-

чаемые в результате побочного эффекта процедуры главной выборки. Последовательность чисел  $\{U_i\}$  будет периодической с периодом  $2^N - 2$ .

Хотя последовательность чисел  $A_1, A_2, \dots, A_n$  не является случайной, она служит хорошим приближением к последовательности случайных чисел, равномерно распределенных на интервале  $[0, 1]$ . Вследствие этого числа, получаемые таким образом, называют *псевдослучайными*.

Рассмотрим встроены процедуры случайного выбора, которые позволяют получать псевдослучайные числа, распределенные по различным законам распределения.

Почти все эти процедуры работают по следующей схеме.

1. Производится обращение к процедуре главной выборки, которая выдает псевдослучайное число из интервала  $[0, 1]$  и изменяет значение своего параметра. Это позволяет получать при последующих обращениях к этой процедуре новые значения ее результата. Для того чтобы дать возможность пользователю получать несколько независимых потоков случайных чисел, одним из формальных параметров всех процедур случайного выбора как раз и является целая переменная, обозначаемая через  $U$ , которая используется в качестве фактического параметра для процедур главной выборки.

2. Выполняется некоторое функциональное преобразование числа, полученного на предыдущем шаге, с целью получения заданного закона распределения.

Все фактические параметры процедур случайного выбора, кроме параметра  $U$ , обозначающего целую переменную, задающую последовательность псевдослучайных чисел, вызываются по значению. Параметр  $U$  вызывается по наименованию, поскольку значение соответствующего ему фактического параметра должно подвергаться изменению при работе процедуры главной выборки. Хотя многие процедуры, рассматриваемые ниже, описаны на симуле-67, все они для повышения эффективности реализованы в коде ЭВМ.

1. **boolean procedure draw (A, U); name U; real A; integer U;**

Эта процедура доставляет значение **true** с вероятностью  $A$  и значение **false** с вероятностью  $(1 - A)$ . При  $A \geq 1$  всегда доставляет значение **true**, а при  $A \leq 0$  — **false**.

Пример 2.11.

Следующий участок программы планирует на системное время 15 100 процессов классов  $A$  и  $B$ . Вероятность выборки класса  $A$  равна 0.75, а класса  $B$  — 0.25.

```

integer I, ПСЧ;
process class A; {.....};
process class B; {.....};
ПСЧ:=3;
for I:=1 step 1 until 100 do
if draw (0.75, ПСЧ) then activate new A at 15
else activate new B at 15;

```

2. **integer procedure randint (A, B, U); name U;**  
**integer A, B, U;**

Доставляет значение, равное одному из целых чисел  $A, A + 1, \dots, B$  с вероятностью  $1/(B - A + 1)$ . Предполагается, что  $B \geq A$ .

3. **real procedure uniform (A, B, U); name U; real A, B;**  
**integer U;**

Значением этой процедуры являются действительные числа, распределенные равномерно на интервале  $[A, B]$  ( $B > A$ ).

**З а м е ч а н и е.** Приведем на примере процедуры **uniform** схему реализации процедуры случайного выбора:

```

real procedure uniform (A, B, U); name U; real A, B;
integer U;
uniform:=A + (B - A) * psrand (U);
4. real procedure normal (A, B, U); name U; real A, B;
integer U;

```

С помощью этой процедуры можно получать псевдослучайные числа, распределенные по нормальному закону с математическим ожиданием  $A$  и среднеквадратическим отклонением  $B$ .

При реализации процедуры применяется аппроксимационная формула для интегральной функции распределения.

5. **real procedure psnorm (A, B, C, U); name U; real A, B;**  
**integer C, U;**

Эта функция также дает нормально распределенные псевдослучайные числа, но для их получения используется сумма  $C$  значений главной выборки, т. е. эта процедура для получения 1 числа многократно изменяет значение переменной, соответствующей параметру  $U$ . Работа процедуры основана на центральной предельной теореме из теории вероятностей.

6. **real procedure negexp (A, U); name U; real A;**  
**integer U;**



Процедура `negexp` доставляет значения, распределенные по экспоненциальному закону со средним  $1/\Lambda$ . Эта процедура часто применяется для моделирования пуассоновского (простейшего) потока событий с интенсивностью  $\Lambda$ .

**Пример 2.12.** Задать генерацию пуассоновского потока процессов класса ЗАЯВКА с интенсивностью  $\Lambda$  штук в единицу времени можно с помощью следующего фрагмента программы ( $\Lambda=5$ ):

```
process class ГЕНЕРАТОР (L, U); integer L, U;  
begin  
  РАБОТА: activate new ЗАЯВКА;  
  hold (negexp (L, U));  
  goto РАБОТА;  
end;  
:  
activate new ГЕНЕРАТОР (5, 3);
```

Если в этой же программе требуется задать еще один поток процессов класса ЗАЯВКА, независимый от первого и имеющий интенсивность 8, то это можно сделать с помощью оператора

```
activate new ГЕНЕРАТОР (8, 7);
```

Моменты времени, в которые будут появляться процессы класса ЗАЯВКА от этого процесса класса ГЕНЕРАТОР, не будут связаны с моментами генерации заявок предыдущим генератором, так как процедура `negexp` в этих процессах использует разные начальные значения параметра (3 в первом случае и 7 во втором).

```
7. integer procedure poisson (Λ, U); name U; real Λ;  
integer U;
```

Значениями этой функции являются целые числа, распределенные по закону Пуассона с параметром  $\Lambda$ .

```
8. integer procedure erlang (L, K, U); name U; real L, K;  
integer U;
```

Процедура доставляет значения, распределенные по закону Эрланга с математическим ожиданием  $1/L$ . Значение процедуры вычисляется как некоторая функция от `entier (K)` значений главной выборки ( $K, L > 0$ ).

Следующие 3 встроенные функции дают возможность генерировать псевдослучайные числа с произвольным законом распределения, который задается тем или иным способом самим пользователем.

9. integer procedure discrete (A, U); name U; array A;  
integer U;

Функция `discrete` доставляет целое значение, лежащее в интервале  $[НГ, ВГ + 1]$  ( $НГ, ВГ$  — нижняя и верхняя границы индекса одномерного фактического массива, соответствующего параметру  $A$ ), равное минимальному  $I$ , для которого  $A[I] > U$ , где  $U$  — значение главной выборки. Если  $A[ВГ] \leq U$ , то результат процедуры полагается равным  $ВГ + 1$ . Иными словами, функция `discrete` трактует массив  $A$  как требуемую интегральную функцию распределения, аргументом которой является индекс.

10. real procedure linear (A, B, U); name U; array A, B;  
integer U;

Как и предыдущая, эта функция позволяет получать псевдослучайные числа, распределенные по закону, задаваемому

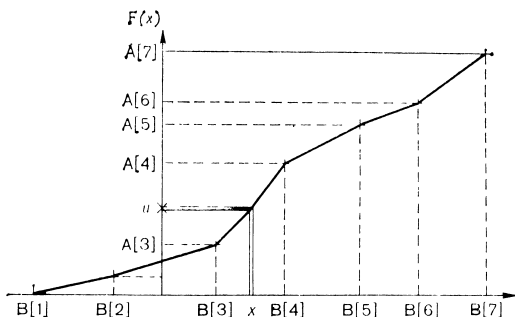


Рис. 17. Работа функции `linear`.

пользователем. Закон распределения определяется с помощью линейной интерполяции неравношаговой таблицы значений интегральной функции распределения, задаваемой массивами  $A$  и  $B$  так, что  $A[I] = F(B[I])$ , где  $A$  и  $B$  — одномерные массивы одинаковой длины, причем значение первого элемента массива  $A$  равно 0, а последнего — 1 и  $A[I] \geq A[J]$ , а  $B[I] > B[J]$  при  $I > J$  (см. рис. 17).

Работа процедуры состоит в получении с помощью процедуры главной выборки значений  $u \in [0, 1]$  и определении соответствующих им значений случайной величины  $X$ , принимаемых за значение функции `linear`. Значения  $X$  будут распределены по закону, заданному пользователем массивами  $A$  и  $B$ .

11. integer procedure histd (A, U); name U; array A;  
integer U;

Эта процедура доставляет целое значение, лежащее в интервале [НГ, ВГ], где НГ и ВГ — нижняя и верхняя границы одномерного массива А, элементы которого определяют относительную частоту вхождения соответствующего индекса. Например, чтобы задать выпадения числа  $-1$  с вероятностью 0.3, числа 0 — с вероятностью 0.4 и числа  $+1$  — с вероятностью 0.3, нужно определить массив, скажем, ГИСТ, состоящий из трех элементов:

**array ГИСТ  $[-1 : +1]$ ,**

задать его элементам соответствующие значения: ГИСТ  $[-1] := 0.3$ ; ГИСТ  $[0] := 0.4$ ; ГИСТ  $[1] := 0.3$ ; и обратиться к процедуре histd:

**for I:=1 step 1 until 100 do X[I]:=histd (ГИСТ, U);**

После выполнения этого оператора цикла в массиве X появятся в случайном порядке 100 значений, равных  $-1$ ,  $+1$  и 0, количество которых определяется заданными для каждого из них вероятностями.

Хотя эта процедура работает медленнее, чем linear, она более удобна, особенно если закон распределения случайных чисел меняется в процессе работы программы.

## 2.5. Процедуры для сбора статистики

Для облегчения программирования типовых приемов статистического анализа результатов моделирования в языке симула-67 имеются две встроенные процедуры, позволяющие накапливать гистограммы значений случайных величин и вести в ходе работы модели интегрирование по системному времени.

Для накопления гистограмм служит процедура

**histo (A, B, C, D);**

где А, В — одномерные массивы арифметического типа, задающие гистограмму; С — наблюдаемое значение некоторой арифметической величины, для которой строится гистограмма (аргумент гистограммы); D — арифметическая величина, равная весу значения С (чаще всего  $C=1$ ).

Значения элементов массива В трактуются процедурой histo как правые границы интервалов аргумента гистограммы, а в массиве А накапливаются данные о попадании исследуемой величины в заданные интервалы. Значение последнего элемента массива А используется для указания количества наблюдений, при которых значения аргумента превышают максимальную границу, заданную последним элементом массива В.

Встроенная процедура *accum* (A, B, C, D), определенная в классе *simulation*, может использоваться для накопления в переменной A интеграла по системному времени от переменной C. Интегрирование по системному времени часто применяется при подсчете средних (по времени) значений параметров моделируемых систем. Параметр B используется для фиксации момента времени, когда значения A и C обновлялись последний раз, а формальный параметр D, которому в качестве фактического может соответствовать произвольное арифметическое выражение, трактуется как приращение переменной C со времени ее предыдущего изменения. Параметры A, B, C вызываются по наименованию и их значения подвергаются изменению при работе процедуры *accum*, поэтому соответствующие им фактические параметры должны быть переменными. Параметр D вызывается по значению.

Процедура *accum* ведет интегрирование методом прямоугольников и легко описывается на языке симула-67:

```
procedure accum (A, B, C, D); name A, B, C; real A, B, C, D;  
  begin A:=A + C*(time — B); B:=time; C:=C + D;  
  end accum;
```

В качестве примера практического использования процедур *histo* и *accum* на рис. 19 приведена декларация класса QSTAT, объекты которого можно применять для сбора статистических данных об очередях. Атрибуты класса QSTAT обеспечивают формирование и хранение следующих данных:

- максимальной длины очереди, наблюдавшейся за время существования объекта класса QSTAT;
- среднего по времени значения длины очереди;
- гистограммы распределения длины очереди.

Для того чтобы воспользоваться предлагаемыми в классе QSTAT средствами, например, для сбора статистики об очереди QUEUE, заданной описанием *ref* (head) QUEUE, достаточно создать соответствующий объект с помощью следующего фрагмента симула-программы:

**Пример 2.14.**

```
ref (QSTAT) QS;  
QS:—new QSTAT ('QUEUE', 5, 10, 8);
```

и поместить после операторов, изменяющих содержимое очереди QUEUE обращения к процедурам класса QSTAT, фиксирующим моменты изменения очереди и выполняющим действия по обновлению накопленных статистических данных. Например, при включении некоторого объекта X в очередь QUEUE при помощи оператора X.*into* (QUEUE) вслед за этим оператором

**class QSTAT** (NAME, A, W, N); **text** NAME; **integer** A, W, N;  
*comment* NAME — имя очереди, A — правая граница крайнего ле-  
 вого интервала, W — ширина интервала, N — количество интерва-  
 лов, включая крайние полубесконечные интервалы;

```
begin integer array H [1 : N], G [1 : N-1];
  real TIMEINT, TSTART, LMEAN, TCHANGE;
  integer LMAX, CURRLEN, I;
  TCHANGE:=TSTART:=TIME;
for I:=1 step 1 until N-1 do
    G[I]:=A * W * (I-1);
  procedure IN; UPDATE (1);
  procedure OUT; UPDATE (-1);
  procedure UPDATE (N); integer N;
    begin
      accum (TIMEINT, TCHANGE, CURRLEN, N);
      if time>TSTART then LMEAN:=TIMEINT/(time-TSTART);
      histo (H, G, GURLEN, 1);
    end UPDATE;
  procedure HPRINT;
    begin outtext (' DISTRIBUTION FOR ');
      outtext (NAME); outimage;
    outtext (' UPPER BOUND NUMBER OF OBSERVATIONS ');
      outimage;
    for I:=1 step 1 until N-1 do
      begin outint (N[I], 11); outint (A[I], 11);
        outimage;
      end;
    outtext (' OVERFLOW '); outint (H[N], 11);
    outimage;
  end HPRINT;
end QSTAT;
```

нужно поместить оператор QS.IN, а при его исключении из очереди после оператора X.out должен исполняться оператор QS.OUT.

Целые числа в фактических параметрах генератора объектов в примере 2.14 задают интервалы значений, которые будут использоваться при формировании гистограммы: 5 — правая граница левого бесконечного интервала, 10 — ширина интервала; 8 — общее количество интервалов, включая крайние бесконечные интервалы.

Обращаться к статистическим данным об очереди QUEUE, накопленным в объекте QS, можно с помощью дистанционных идентификаторов: QS.LMAX даст максимальную длину очереди, QS.LMEAN — среднюю по времени длину очереди. Оператор QS.HPRINT напечатает гистограмму распределения длины очереди QUEUE в следующем виде:

#### DISTRIBUTION FOR QUEUE;

UPPER BOUND	NUMBER OF OBSERVATIONS
5	18
10	40
15	48
20	60
25	100
30	70
35	55
OVERFLOW	44

В главе 5, а также в работах [38, 42] можно найти более развитые средства сбора статистики, построенные с использованием процедур histo и assum, а также других средств языка.

## 2.6. Примеры программ моделирования на языке симула-67

### 2.6.1. Модель станции технического обслуживания автомобилей.

Рассмотрим применение языка симула-67 для моделирования работы систем массового обслуживания на примере простой модели станции технического обслуживания автомобилей. Пусть моделируемая станция имеет K мест для стоянки автомобилей, ожидающих обслуживания, и одну бригаду рабочих, которые выполняют необходимые технологические операции. Бригада обрабатывает автомобили в порядке поступления, причем, обработка следующего автомобиля начинается только после завершения обработки предыдущего. Допустим, что время обработки одного автомобиля — случайная величина, распреде-

ленная по нормальному закону с математическим ожиданием  $M$  и среднеквадратическим отклонением  $S$ .

Будем считать, что автомобили, прибывающие на станцию технического обслуживания, образуют пуассоновский поток с интенсивностью  $A$  штук в единицу времени, и автомобиль становится в очередь на обслуживание только в том случае, если на стоянке имеется хотя бы одно свободное место.

Целью моделирования будем считать получение следующих характеристик работы станции технического обслуживания:

- общее количество автомобилей, прибывавших на станцию ( $KA$ );

- количество обслуженных автомобилей ( $KOA$ );

- количество автомобилей, поступивших на обслуживание сразу же после прибытия на станцию ( $HEJD$ );

- среднее время ожидания обслуживания для тех автомобилей, которым пришлось стоять в очереди ( $CBOЖ$ );

- загрузка бригады ( $ЗБ$ ), вычисляемая как отношение времени, которое затрачено на обслуживание автомобилей, к общему времени работы станции.

Работу станции будем моделировать в интервале времени от 0 до  $T_M$ , считая, что первый автомобиль появляется в нулевой момент времени.

При программировании имитационных моделей на языке симула-67 удобно сначала описать в виде деклараций классов сценарии поведения объектов, составляющих моделируемую систему, а затем задать программное окружение, в котором определяются условия проведения имитационного эксперимента и создаются объекты, отображающие начальную конфигурацию системы.

В примере 2.15 приведены тексты деклараций классов АВТОМОБИЛЬ и БРИГАДА, описывающих поведение автомобилей, поступающих на станцию, и процесс их обслуживания. Вслед за примером даются пояснения к каждой строке текста.

#### Пример 2.15

1. process class АВТОМОБИЛЬ;
2. begin real HO;
3. KA:=KA + 1;
4. if KЗМ=K then goto КОНЕЦ;
5. HO:=time; this АВТОМОБИЛЬ.into (СТОЯНКА);
6. KЗМ:=KЗМ + 1;
7. activate БРИГ after current;
8. КОНЕЦ: activate new АВТОМОБИЛЬ delay negexp (A, U1);
9. end АВТОМОБИЛЬ;

```

10. process class БРИГАДА;
11.   begin real НРАБ, ТРАБ, ЖДАЛ;
      ref (АВТОМОБИЛЬ) КЛИЕНТ;
12.   РАБОТА: if СТОЯНКА.empty then passivate;
13.   НРАБ:=time; КЛИЕНТ:—СТОЯНКА.first; КЛИЕНТ.out;
14.   ЖДАЛ:=time — КЛИЕНТ.НО;
15.   if ЖДАЛ=0 then НЕЖД:=НЕЖД + 1 else
      ТОЖ:=ТОЖ + ЖДАЛ;
16.   КЗМ:=КЗМ — 1;
17.   hold (normal(M, S, U2));
18.   ТРАБ:=ТРАБ + (time — НРАБ);
19.   КОА:=КОА + 1; goto РАБОТА;
20. end БРИГАДА;

```

Строка 1. Начало декларации класса АВТОМОБИЛЬ.

Строка 2. Начало блока тела декларации класса АВТОМОБИЛЬ. Описание атрибута НО, предназначенного для хранения времени начала ожидания обслуживания для автомобилей, прибывающих на станцию.

Строка 3. Автомобиль, прибывший на станцию, увеличивает на 1 общее количество автомобилей (переменная КА).

Строка 4. Если на стоянке для обслуживания заняты все места, то автомобиль покидает станцию, запланировав появление следующего автомобиля.

Строка 5. Автомобиль фиксирует время начала своего ожидания и встает в очередь на обслуживание.

Строка 6. Количество занятых мест (переменная КЗМ) увеличивается на 1.

Строка 7. Планирование начала работы бригады на текущий момент времени, если она простаивала в момент прибытия автомобиля.

Строка 8. Порождение следующего автомобиля и планирование начала его работы через случайный интервал времени, распределенный по экспоненциальному закону со средним значением  $1/A$ . (Автомобили образуют пуассоновский поток с интенсивностью  $A$  штук в единицу времени.)

Строка 9. Конец декларации класса АВТОМОБИЛЬ. При прохождении управления через **end АВТОМОБИЛЬ** процесс переходит в завершенное состояние. Процессы, отображающие автомобили, покидающие станцию, при этом уничтожаются, поскольку на них нет ссылок.

Строка 10. Начало декларации класса БРИГАДА.

Строка 11. Описания переменных, служащих атрибутами бригады: НРАБ — время начала работы по обслуживанию очередного автомобиля; ТРАБ — суммарное время работы бригады, исключая время простоев из-за отсутствия клиентов; ЖДАЛ —



время, которое очередной автомобиль ждал начала обслуживания; КЛИЕНТ — ссылка на обслуживаемый в данный момент автомобиль.

Строка 12. Начало работы бригады. Если на стоянке нет автомобилей, ожидающих обслуживания, то бригада простаивает. Простой отображается пассивным состоянием процесса класса БРИГАДА.

Строка 13. Начало обслуживания очередного автомобиля: в переменной НРАБ отмечается время начала работы с очередным клиентом; переменная КЛИЕНТ устанавливается на первый автомобиль на стоянке; этот автомобиль исключается из очереди.

Строка 14. Вычисляется время, которое клиент потратил на ожидание в очереди.

Строка 15. Если очередной клиент не ждал обслуживания, то увеличивается на 1 значение переменной НЕЖД, в противном случае обновляется значение глобальной переменной ТОЖ, в которой накапливается суммарное время ожидания клиентов.

Строка 16. Количество мест, занятых на стоянке для ожидания обслуживания, уменьшается на 1.

Строка 17. Имитация обработки автомобиля: процесс класса БРИГАДА задерживается на случайное время, распределенное по нормальному закону с математическим ожиданием  $M$  и среднеквадратическим отклонением  $S$ .

Строка 18. Увеличение суммарного времени работы бригады на время обработки очередного клиента.

Строка 19. Увеличение на 1 количества обслуженных автомобилей и переход на обработку следующего клиента.

Строка 20. Конец декларации класса БРИГАДА. Для завершения разработки модели станции технического обслуживания необходимо охватить приведенные декларации классов блоками, в которых задаются глобальные переменные, определяющие параметры моделируемой системы, инициализируется работа модели и выводятся результаты моделирования. В примере 2.16 приводится текст симула-программы, обеспечивающей моделирование станции обслуживания, если время ее работы, параметры станции обслуживания, а также интенсивность потока автомобилей задаются в качестве входных данных.

Пример 2.16.

1. **begin real** TM, ТОЖ, A, ЗБ, M, S;  
    **integer** K, KЗМ, KA, KOA, НЕЖД, U1, U2;
2.     TM:=inreal; M:=inreal; S:=inreal; A:=inreal;  
       K:= inint;
3.     outtext ('МОДЕЛИРОВАНИЕ СТАНЦИИ  
              ТЕХНИЧЕСКОГО'); outimage;

```

4.  outtext ('ОБСЛУЖИВАНИЯ АВТОМОБИЛЕЙ:'); outimage;
5.  outtext ('ВРЕМЯ РАБОТЫ:'); outfix (ТМ, 2, 10);
    outimage;
6.  outtext ('ИНТЕНСИВНОСТЬ ПОТОКА:'); outfix (А, 2, 5);
    outimage;
7.  outtext ('ПАРАМЕТРЫ ОБСЛУЖИВАНИЯ:'); outimage;
8.  outtext ('M='); outfix (M, 2, 8);
9.  outtext ('S='); outfix (S, 2, 8);
10. outtext ('K='); outint (K, 5); outimage;
11. simulation begin ref (БРИГАДА) БРИГ;
    ref (HEAD) СТОЯНКА;
12.   process class АВТОМОБИЛЬ; ...;
13.   process class БРИГАДА; ...;
14.   U1:=1; U2:=3; КЗМ:=КА:=КОА:=НЕЖД:=0;
15.   СТОЯНКА:—new head;
16.   БРИГ:—new БРИГАДА;
17.   activate new АВТОМОБИЛЬ;
18.   hold (ТМ);
19.   ЗБ:=БРИГ. ТРАБ/ТМ; end блока simulation;
20.   if НЕЖД < КОА then СВОЖ:=ТОЖ/(КОА — НЕЖД);
21.   outtext ('РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ:');
    outimage;
22.   outtext ('ПРИБЫЛО АВТОМОБИЛЕЙ:'); outint (КА, 6);
    outimage
23.   outtext ('ИЗ НИХ ОБСЛУЖЕНО:'); outint (КОА, 6);
24.   outtext ('НЕ ЖДАЛИ:'); outint (НЕЖД, 6); outimage;
25.   outtext ('СРЕДНЕЕ ВРЕМЯ ОЖИДАНИЯ:');
    outfix (СВОЖ, 2, 10); outimage;
26.   outtext ('ЗАГРУЗКА БРИГАДЫ:'); outfix (ЗБ, 3, 5);
27. end

```

Пояснения к примеру 2.16.

Строка 1. Начало симула-программы. Описания глобальных переменных, используемых для задания параметров моделируемой станции, интенсивности потока автомобилей и для вычисления результатов моделирования.

Строка 2. Ввод исходных данных, необходимых для работы модели. Ввод производится с системного устройства ввода, данные должны содержать 4 вещественных константы и одну целую константу, разделенные пробелами или другими символами, которые не могут встречаться в записи числа. Например, для задания вводимых данных может быть использована перфокарта со следующей информацией:

ВРЕМЯ=1440, ОБСЛУЖИВАНИЕ 10.5, 1.35  
ИНТЕНСИВ: 0.5; МЕСТ 6

(более подробно средства ввода-вывода рассматриваются в главе 3).

Строки 3—10. Печать введенных исходных данных. В результате исполнения приведенных операторов вывода будет напечатана (выведена на системное устройство печати) следующая информация (предполагается, что  $TM = 1440$ ,  $M = 10.5$ ,  $S = 1.35$ ,  $A = 0.5$ ,  $K = 6$ ):

**МОДЕЛИРОВАНИЕ СТАНЦИИ ТЕХНИЧЕСКОГО  
ОБСЛУЖИВАНИЯ АВТОМОБИЛЕЙ.**

**ВРЕМЯ РАБОТЫ: 1440.00**

**ИНТЕНСИВНОСТЬ ПОТОКА: 0.50**

**ПАРАМЕТРЫ ОБСЛУЖИВАНИЯ:**

**$M = 10.50$   $S = 1.35$   $K = 6$**

Строка 11. Начало блока с префиксом *simulation*. Описание ссылочной переменной БРИГ, которая будет обозначать процесс класса БРИГАДА, отображающий бригаду рабочих станции технического обслуживания, и переменной СТОЯНКА для ссылки на голову набора, представляющего стоянку автомобилей, ожидающих обработки.

Строки 12—13. На их место должны быть помещены декларации классов из примера 2.15.

Строка 14. Присваивание начальных значений целым переменным U1, U2, используемым при получении псевдослучайных чисел, и переменных КА, КОА, НЕЖД.

Строка 15. Создание головы набора СТОЯНКА.

Строка 16. Генерация процесса класса БРИГАДА и присваивание ему имени БРИГ.

Строка 17. Создание и запуск в работу первого автомобиля, который, в свою очередь, активизирует процесс БРИГ (см. строку 7 в примере 2.15).

Строка 18. Процесс Главная программа задерживается на время TM, в течение которого будут взаимодействовать процессы класса АВТОМОБИЛЬ и процесс БРИГ, имитируя работу станции технического обслуживания.

Строка 19. Вычисление загрузки бригады. Моделирование прекращается, т. к. управление покидает блок с префиксом *simulation*.

Строка 20. Обработка результатов моделирования. Вычисляются среднее время ожидания обслуживания для тех автомобилей, которые были задержаны на стоянке.

Строки 21—26. Вывод результатов моделирования. После исполнения приведенных операторов будет выведена следующая информация:

**РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ:**  
**ПРИБЫЛО АВТОМОБИЛЕЙ: 1785**  
**ИЗ НИХ ОБСЛУЖЕНО: 181, НЕ ЖДАЛИ: 17**  
**СРЕДНЕЕ ВРЕМЯ ОЖИДАНИЯ: 30.45**  
**ЗАГРУЗКА БРИГАДЫ: 0.98**

Строка 27. Конец симула-программы. Возврат управления в операционную систему.

Рассмотрим, как изменится программа примера 2.16, если в результате моделирования нужно определить, при какой интенсивности потока автомобилей доля обслуженных машин превысит 80%, если характеристики станции обслуживания остаются неизменными. При решении указанной задачи методом имитационного моделирования нужно многократно проводить прогоны модели станции, постепенно уменьшая интенсивность потока автомобилей. Будем считать, что нам достаточна 10%-я точность определения искомой интенсивности, и во входных данных задается ее исходное значение, при котором доля обслуженных машин заведомо меньше 80%.

При этих условиях поиск требуемого значения интенсивности потока обеспечивается следующими дополнениями в симула-программу примера 2.16:

— блок с префиксом *simulation*, исполнение которого имитирует работу станции обслуживания, помечается некоторой меткой, например ПРОГОН, т. е. строка 11 в примере 2.16 принимает вид

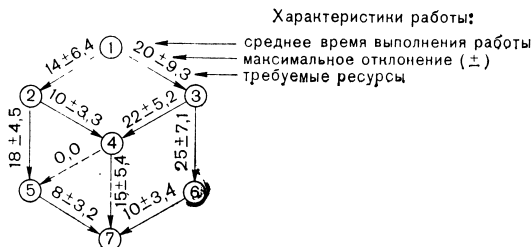
**ПРОГОН: simulation begin ... <как и раньше>;**

— после строки 20 вставляются операторы, которые уменьшают на 10% текущее значение интенсивности, если не достигнут требуемый процент обслуживания автомобилей, и передают после этого управление на метку ПРОГОН для повторения имитационного эксперимента. Если процент обслуженных автомобилей больше 80, то найденное значение интенсивности печатается, после чего выдаются остальные результаты моделирования:

```
if KOA/KA < 0.8 then
begin A:=0.9*A; goto ПРОГОН end;
outtext ('ИСКАМАЯ ИНТЕНСИВНОСТЬ='); outfix (A, 2, 10);
outimage;
```

**2.6.2. Модель выполнения работ по сетевым графикам.** Рассмотрим применение метода имитационного моделирования для решения задачи определения срока заверше-

ния комплекса взаимосвязанных работ, выполняемых в соответствии с сетевым графиком. Постановка задачи и исходные данные для конкретного сетевого графика взяты из примера 7D, приведенного в книге Т. Дж. Шрайбера «Моделирование на



Работа	Ресурсы	Среднее время	+	-
1 2	4	14	6	
1 3	3	20	9	
2 5	5	18	4	
2 4	3	10	3	
3 4	2	22	5	
3 6	1	25	7	
3 5	0	0	0	
4 7	4	15	5	
5 7	2	8	3	
6 7	4	10	3	

Рис. 20. Сетевой график и его числовое представление.

GPSS» [31], где дается решение этой задачи с помощью программы на языке GPSS. Вид этой программы целиком определяется топологией конкретного сетевого графика, который требуется исследовать. На языке симула-67 можно написать программу, для которой исследуемый сетевой график будет задаваться входными данными, что позволяет многократно использовать ее для моделирования различных графиков.

Пусть требуется определить среднее время выполнения комплекса работ, задаваемого сетевым графиком, при условии, что каждая работа требует определенное количество ресурсов и длится случайное время. Следуя работе [31], будем считать ресурсы одноименными, а время выполнения работ — распределенным по равномерному закону в интервале  $[T - D, T + D]$ , где  $T$  — среднее время, а  $D$  — максимально возможное отклонение, задаваемые для каждой работы. Относительно ресурсов, требуемых для каждой работы, предполагается, что по окончании работы они освобождаются и могут быть снова использованы. На рис. 20 изображен сетевой график, исследуемый GPSS — программой примера 7D из книги Т. Дж. Шрайбера [31], и дано

его числовое представление, используемое при моделировании этого графика с помощью симула-программы.

Решение поставленной задачи удобно представить в виде взаимодействия ряда процессов, отображающих выполнение работ, задаваемых сетевым графиком. Процессы, отображающие узлы сетевого графика, запускаются в работу в те моменты времени, когда завершаются все работы, входящие в данный узел. В процессе своего функционирования узел должен инициировать исполнение всех исходящих из него работ, для которых имеется достаточное количество ресурсов, а также тем или иным способом зафиксировать работы, которые не могут быть начаты из-за нехватки ресурсов.

В задачу процессов, отображающих выполнение работ в сетевом графике, входит имитация требуемого ей случайного интервала времени, по истечении которого нужно возвратить ресурсы, возможно запустив при этом некоторые из работ, задержанных по нехватке ресурсов. Кроме того, при своем завершении работа должна запустить процесс, отображающий тот узел, в который она входит, если все остальные входящие в него работы уже окончены.

Для определения времени выполнения комплекса работ достаточно вычислить разность времен начала функционирования исходного и завершающего узлов сетевого графика. Чтобы получить достоверные оценки среднего времени выполнения работ, нужно многократно повторить имитацию хода работ, обеспечив случайный характер времени каждой работы.

Симула-программа моделирования сетевого графика состоит из деклараций классов для процессов, соответствующих узлам и работам, и операторов, обеспечивающих их генерацию, многократное исполнение модели, а также подсчет и вывод среднего значения времени завершения работ.

В примере 2.17 приведено описание правил действий объектов, отображающих узлы сетевого графика, в виде декларации класса УЗЕЛ. Вслед за примером даются подробные пояснения к тексту декларации.

**Пример 2.17.**

1. process class УЗЕЛ (НОМЕР, ДУГИ); integer НОМЕР;  
ref (head) ДУГИ;
2. begin integer ВХР, КЗР; ref (РАБОТА) P;
3. НАЧАЛО: if ДУГИ.empty then begin
4. activate ГП at time; goto ЖДУ; end работы УЗЛА;
5. for P:—ДУГИ.first, P.suc while P≠none do
6. if P.ПОТР>НАЛИЧИЕ then  
new НЕХВАТКА (P).into (ЖДУЩИЕ)

7.       **else begin НАЛИЧИЕ:=НАЛИЧИЕ—Р.ПОТР;**  
           **activate P delay 0 end;**
8.       **ЖДУ: КЗР:=0; passivate; goto НАЧАЛО;**
9.       **end УЗЕЛ;**

Строка 1. Заголовок декларации класса УЗЕЛ. Параметр **НОМЕР** введен для идентификации конкретного узла, а в набор **ДУГИ** при генерации совокупности объектов, отображающих сетевой график, заносятся объекты класса **РАБОТА**, которые представляют работы, исходящие из данного узла.

Строка 2. Начало тела класса УЗЕЛ. В атрибут **ВХР** при генерации сетевого графика заносится количество работ, входящих в данный узел, а значение переменной **КЗР** указывает число входящих работ, завершенных к текущему моменту времени. Ссылочная переменная **Р** используется для просмотра набора **ДУГИ** при запуске работ, исходящих из данного узла.

Строка 3. Начало работы узла. Поскольку исходящих дуг не имеет только завершающий узел сетевого графика, то истинность выражения **ДУГИ.empty** означает, что активен завершающий узел, т. е. выполнены все входящие в него работы, и можно выдавать результаты моделирования.

Строка 4. Планирование активной фазы для процесса Главная программа (на него ссылается глобальная переменная **ГП**), который вычисляет время выполнения комплекса работ и вновь запускает моделирование сетевого графика, если не достигнуто требуемое число прогонов модели. После указанного планирования производится передача управления на участок программы, обеспечивающий подготовку узла к следующей активной фазе.

Строка 5. Заголовок цикла, обеспечивающий применение следующего за ним оператора ко всем членам набора **ДУГИ**. На каждом шаге цикла переменная **Р** указывает на очередную исходящую работу.

Строка 6. Если потребность в ресурсах у очередной работы (атрибут **ПОТР** у объектов класса **РАБОТА**) превышает наличные ресурсы (значение глобальной переменной **НАЛИЧИЕ**), то эта работа не запускается, а в набор **ЖДУЩИЕ** вносится объект класса **НЕХВАТКА**, ссылающийся на задержанную работу.

Строка 7. Для очередной работы имеется достаточное количество ресурсов. Значение переменной **НАЛИЧИЕ** уменьшается на размер ресурсов, выделенных этой работе, и с помощью оператора **activate P delay 0** обеспечивается ее начало, но после того, как узел закончит просмотр всех исходящих работ. Порядок следования процессов класса **РАБОТА** в наборе **ДУГИ** соответствует порядку запуска работ, исходящих из данного

узла, и очередности предоставления ресурсов. Из нескольких работ с одинаковой потребностью в ресурсах раньше начнет исполняться та работа, которая стоит ближе к началу набора ДУГИ. Порядок занесения работ в этот набор совпадает с очередностью следования данных о работах, исходящих из одного узла. Например, при наличии 5 и более единиц наличных ресурсов в момент работы узла 2 (см. рис. 20) первым будет запущен процесс, соответствующий работе 2—5, поскольку во входных данных она представлена раньше работы 2—4. Однако при наличии 4 или 3 единиц ресурсов первой начнется выполнение работы 2—4, так как для работы 2—5 не хватит ресурсов.

Строка 8. Завершение работы узла и подготовка к работе во время следующего прогона модели: количество завершенных работ, входящих в данный узел, обнуляется; текущий процесс переводится в пассивное состояние; оператор НАЧАЛО обеспечит повторное исполнение правил действий узла после его активизации на следующем прогоне.

Строка 9. Конец декларации класса УЗЕЛ.

Пример 2.18 содержит декларацию класса РАБОТА, определяющего процессы, функционирование которых отображает выполнение работ в сетевом графике, а также описание вспомогательного класса НЕХВАТКА, используемого при имитации задержки работ из-за нехватки ресурсов.

Пример 2.18.

1. process class РАБОТА (Т, D, ПОТР, ЦЕЛЬ, U); real Т, D;  
integer ПОТР, U; ref (УЗЕЛ) ЦЕЛЬ;
2. begin ref (НЕХВАТКА) Н, R;
3. ИСПОЛНЕНИЕ: hold (uniform (Т — D, Т + D, U));
4. НАЛИЧИЕ := НАЛИЧИЕ + ПОТР;
5. if ЖДУЩИЕ.empty then goto ЗАПУСК;
6. R:—ЖДУЩИЕ.first; for Н:—R while Н≠/none do
7. if Н.КОМУ.ПОТР ≤ НАЛИЧИЕ then  
begin НАЛИЧИЕ := НАЛИЧИЕ + Н.КОМУ.ПОТР;
8. R:—Н.suc; Н.out; activate Н.КОМУ delay 0;
9. end else R:—Н.suc;
10. ЗАПУСК: ЦЕЛЬ.КЗР := ЦЕЛЬ.КЗР + 1;
11. if ЦЕЛЬ.КЗР = ЦЕЛЬ.ВХР then activate ЦЕЛЬ delay 0;
12. passivate; goto ИСПОЛНЕНИЕ  
end РАБОТА;
13. link class НЕХВАТКА (КОМУ); ref (РАБОТА) КОМУ; ;

Дадим подробные пояснения к текстам деклараций, приведенных в примере 2.18.

Строка 1. Заголовок декларации класса РАБОТА. Определяются атрибуты, характерные для каждой работы в сетевом гра-



фике:  $T$  — среднее время выполнения работы,  $D$  — максимально возможное отклонение от среднего времени, ПОТР — потребность в ресурсах для выполнения данной работы, ЦЕЛЬ — ссылка на процесс класса УЗЕЛ, соответствующий тому узлу сетевого графика, в который входит работа,  $U$  — целая переменная, используемая при получении псевдослучайных чисел.

Наличие атрибута  $U$  у каждого процесса класса РАБОТА обеспечивает взаимную независимость псевдослучайных чисел, разыгрываемых при определении времени работ в сетевом графике. Заметим, что в GPSS — программе (см. пример 7D в книге [31]), имитирующей сетевой график, изображенный на рис. 20, длительности всех работ вычисляются при помощи одного датчика случайных чисел, а это ведет к взаимной зависимости времен выполнения различных работ.

Строка 2. Начало тела класса РАБОТА. Описываются рабочие ссылочные переменные  $H$  и  $R$ , используемые при просмотре списка работ, задержанных из-за нехватки ресурсов.

Строка 3. Имитация выполнения работы в течение случайного интервала времени, распределенного равномерно в интервале  $[T - D, T + D]$ .

Строка 4. Выполнение работы закончено. Производится возврат ресурсов, которыми она пользовалась.

Строка 5. Если пуст набор ЖДУЩИЕ, содержащий объекты класса НЕХВАТКА со ссылками на задержанные из-за нехватки ресурсов работы, то законченная работа пытается активизировать тот узел, в который она входит. Соответствующий участок декларации класса РАБОТА помечен меткой ЗАПУСК.

Строка 6. Переменная  $R$  устанавливается на первый член набора ЖДУЩИЕ, после чего начинается просмотр этого набора. Переменная  $H$  указывает на его очередной элемент.

Строки 7—8. Если потребность в ресурсах для очередной работы, представленной в наборе ЖДУЩИЕ объектом  $H$  класса НЕХВАТКА, не превышает наличных ресурсов, то для этой работы планируется начало выполнения на текущий момент времени. Предварительно уменьшается количество наличных ресурсов, переменная  $R$  устанавливается на следующий за  $H$  член набора ЖДУЩИЕ, а объект  $H$  исключается из этого набора.

Строка 9. Для задержанной работы, на которую посредством атрибута КОМУ ссылается объект  $H$ , не хватает наличных ресурсов. Объект  $H$  остается в наборе ЖДУЩИЕ, соответствующая работа не запускается, а переменная  $R$  устанавливается на следующий член набора.

Строка 10. Выполненная работа увеличивает на 1 значение атрибута КЗР (количество завершенных работ) у узла-цели, т. е. того узла, в который она входит.

Строка 11. Если выполненная работа оказалась для узла-цели последней, завершения которой он ждал, то работа планирует активную фазу, для этого узла на текущий момент времени, чтобы он мог инициировать исполнение исходящих из него работ. Условием возможности начала работы узла является равенство количества входящих в него работ (атрибут ВХР) и количества завершенных работ (атрибут КЗР).

Строка 12. Функции работы исчерпаны, она переходит в пассивное состояние. Оператор **goto** ИСПОЛНЕНИЕ обеспечит повторение правил действий на следующем прогоне модели.

Строка 13. Описание вспомогательного класса НЕХВАТКА. Атрибут КОМУ указывает на работу, задержанную из-за нехватки ресурсов. Никаких действий объектам класса НЕХВАТКА не предписывается: его тело является пустым оператором, поэтому вслед за спецификацией параметра КОМУ следует символ ; (точка с запятой), завершающий эту декларацию.

Нам остается описать программное окружение для рассмотренных деклараций классов, которое обеспечивает ввод исходных данных, генерацию и запуск в работу процессов, отображающих узлы и работы сетевого графика, вывод результатов моделирования.

Пример 2.19.

1. **begin integer** НУЗЛ, НДУГ, ПЕСМИН, ПЕСМАХ, ПЕСУРС, НАЛИЧИЕ, N, I, K;
2. ПЕСМИН:=inint; ПЕСМАХ:=inint; N:=inint;
3. НУЗЛ:=inint; НДУГ:=inint;
4. outtext ('ИССЛЕДОВАНИЕ СЕТЕВОГО ГРАФИКА ИЗ');  
outint (НУЗЛ, 4);
5. outtext ('УЗЛОВ И'); outimage outint (НДУГ, 5);
6. outtext ('РАБОТ'); outimage;
7. outtext ('ДИАПАЗОН РЕСУРСОВ ОТ'); outint (ПЕСМИН, 5);
8. outtext ('ДО'); outint (ПЕСМАХ, 5); outimage;
9. outtext ('ЧИСЛО ИСПЫТАНИЙ ='); outint (N, 5);  
outimage;
10. simulation **begin ref** (process) ГП;  
**ref** (УЗЕЛ) array У[1:НУЗЛ];
11. **ref** (head) ЖДУЩИЕ; **real** ТСП, ОТКЛ, ТНАЧ, TS;  
**integer** А, В, ПЕС;
12. process class УЗЕЛ ... (см. пример 2.17);
13. process class РАБОТА ... (см. пример 2.18);
14. link class НЕХВАТКА ... (см. пример 2.18);
15. ЖДУЩИЕ:—new head;
16. **for** I:=1 **step** 1 **until** НУЗЛ **do**  
У[I]:—new УЗЕЛ (I, new head);

```

17.  outtext ('ОПИСАНИЕ РАБОТ'); outimage;
18.  outtext ('ОТКУДА КУДА РЕСУРСЫ ВРЕМЯ + -');
    outimage;
19.  for I:=1 step 1 until НДУГ do
20.  begin A:=inint; B:=inint; PEC:=inint; TCP:=inreal;
    ОТКЛ:=inreal;
21.  outint (A, 6); outint (B, 6); outint (PEC, 6);
22.  outfix (TCP, 2, 8); outfix (ОТКЛ, 2, 8); outimage;
23.  new РАБОТА (TCP, ОТКЛ, PEC, У[B], 2*I-1).into
    (У[A].ДУГИ);
24.  У[B].ВХР:=У[B].ВХР+1; end создания работ;
25.  outtext ('РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ'); outimage;
26.  outtext ('РЕСУРСЫ СРОК ВЫПОЛНЕНИЯ'); outimage;
27.  for PECУРС:=РЕСМІN step 1 until РЕСМАХ do
28.  begin outint (РЕСУРС, 5); TS:=0; ГП:—current;
29.  for K:=1 step 1 until N do
30.  begin НАЛИЧИЕ:=РЕСУРС; ТНАЧ:=time;
31.    activate У[1] at time; passivate;
32.    TS:=TS + (time — ТНАЧ); end одного прогона;
33.    outfix (TS/N, 2, 12); outimage;
34.    end N прогонов с одним ресурсом;
35.  end блока simulation;
36. end программы;

```

Пояснения к программе, приведенной в примере 2.19.

Строка 1. Начало симула-программы. Описания глобальных переменных, определяющих условия проведения имитационных экспериментов:

НУЗЛ, НДУГ — количество узлов и дуг (работ) сетевого графика;

РЕСМІN, РЕСМАХ — диапазон значений количества ресурсов, при которых нужно определить среднее время завершения работ;

РЕСУРС — общее количество ресурсов, выделенных для всех работ. Устанавливается перед началом имитации выполнения работ и изменяется от РЕСМІN до РЕСМАХ с шагом 1;

НАЛИЧИЕ — наличное количество свободных ресурсов во время исполнения работ. Начальное значение — РЕСУРС, диапазон изменения — [0, РЕСУРС];

N — количество прогонов (испытаний) модели сетевого графика для определения среднего времени завершения комплекса работ при одном значении количества выделенных ресурсов (переменная РЕСУРС);

I, K — рабочие переменные.

Строки 2—3. Ввод значений для переменных РЕСМІN, РЕСМАХ, N, НУЗЛ, НДУГ. Например, для сетевого графика,

изображенного на рис. 20, можно задать исходные данные для этих переменных в следующей форме, поместив их на одну или несколько перфокарт (записей в наборе данных);

РЕСУРСЫ ОТ 5 ДО 12, N = 250, УЗЛОВ 7, ДУГ 10

Значения диапазона ресурсов и количество испытаний взяты из примера 7D книги [31].

Строки 4—9. Вывод на печать введенных значений. При выполнении приведенных операторов вывода будут напечатаны следующие 4 строки:

ИССЛЕДОВАНИЕ СЕТЕВОГО ГРАФИКА ИЗ 7 УЗЛОВ И  
10 РАБОТ

ДИАПАЗОН РЕСУРСОВ ОТ 5 ДО 12

ЧИСЛО ИСПЫТАНИЙ = 250

Строки 10—11. Начало блока моделирования. Определение переменных, используемых всеми процессами модели:

ГП — ссылка на процесс Главная программа, иницирующий работу модели и выполняющий анализ результатов моделирования;

У[1 : NUЗЛ] — массив ссылок на процессы, отображающие узлы сетевого графика;

ЖДУЩИЕ — набор, в который заносятся объекты класса НЕХВАТКА, ссылающиеся на работы, задержанные из-за нехватки ресурсов;

ТНАЧ — переменная, в которой фиксируется системное время начала очередного прогона модели;

TS — переменная для накопления суммы времен завершения комплекса работ при различных прогонах модели сетевого графика;

ТСР, ОТКЛ, А, В, РЕС — рабочие переменные, используемые при вводе данных о работах сетевого графика.

Строки 12—14. Описания деклараций классов УЗЕЛ, РАБОТА, НЕХВАТКА из примеров 2.17 и 2.18.

Строка 15. Заведение головы набора ЖДУЩИЕ.

Строка 16. Создание NUЗЛ объектов, отображающих узлы сетевого графика. Переменная У[1] указывает на исходный узел графика.

Строки 17—18. Выдача заголовка для печати данных о работах сетевого графика. В результате исполнения этих строк будет напечатано:

ОПИСАНИЕ РАБОТ

ОТКУДА КУДА РЕСУРСЫ ВРЕМЯ + —

Строка 19. Заголовок цикла для ввода данных о работах и генерации процессов класса РАБОТА.

Строка 20. Ввод данных об одной работе. Предполагается, что информация о работах сетевого графика задается в виде НДУГ пятерок чисел, каждая из которых задает одну работу.

Первое число из пятерки вводится в переменную А и означает номер узла, из которого исходит работа. Вслед за ним идет номер узла-цели, вводимый в переменную В. Третье число, равное потребности в ресурсах, вводится в переменную РЕС. Последние два числа задают среднее время выполнения работы (ТСР) и максимально возможное отклонение от среднего (ОТКЛ). На рис. 20 приведено числовое представление работ сетевого графика из примера 7D книги [31], которое можно использовать в качестве входных данных к рассматриваемой симула-программе. Их следует поместить вслед за данными, вводимыми при исполнении строк 2—3.

Строки 21—22. Вывод данных о работах сетевого графика. По окончании цикла под заголовком, напечатанным при исполнении строк 17—18, будут напечатаны НДУГ строчек чисел вида (рис. 20):

1	2	4	14.00	6.00
1	3	3	20.00	9.00
.	.	.	.	.
6	7	4	10.00	3.00

Строка 23. Генерация объекта класса РАБОТА со значениями атрибутов  $T = \text{ТСР}$ ,  $D = \text{ОТКЛ}$ ,  $\text{ПОТР} = \text{РЕС}$ ,  $\text{ЦЕЛЬ} = U[B]$ ,  $U = 2 \times I - 1$  ( $I$ -е нечетное число) и включение созданного объекта в набор ДУГИ процесса  $U[A]$ , отображающего узел, из которого исходит работа.

Строка 24. Увеличение на 1 числа входящих работ у узла-цели. Конец цикла создания работ. Модель сетевого графика полностью подготовлена к работе.

Строки 25—26. Выдача заголовка для печати результатов моделирования.

Строка 27. Заголовок цикла, на каждом шаге которого для исходного значения наличных ресурсов, принадлежащего интервалу  $[\text{РЕСМИН}, \text{РЕСМАХ}]$ , путем  $N$ -кратного прогона модели сетевого графика вычисляется среднее время завершения комплекса работ.

Строка 28. Вывод исходного значения наличных ресурсов перед началом моделирования сетевого графика. Обнуление суммарного времени завершения работ и занесение в переменную ГП ссылки на процесс Главная программа, который в данный момент является текущим.

Строка 29. Заголовок цикла, обеспечивающего  $N$ -кратное повторение прогона модели сетевого графика.

Строка 30. Подготовка к очередному прогону модели: перед началом работ все выделенные ресурсы объявляются свободными, а в переменной ТНАЧ фиксируется время начала работ, исходящих из первого узла сетевого графика.

Строка 31. Планирование активной фазы исходного узла графика на текущий момент системного времени. Запланировав запуск первого узла, Главная программа переходит в пассивное состояние, ожидая «толчка» от завершающего узла сетевого графика.

Строка 32. С этой точки программы возобновляется работа Главной программы после завершения одного прогона модели, т. е. однократной имитации выполнения комплекса работ. Выполняется увеличение суммы времен завершения работ по сетевому графику на время завершения, определенное в результате данного прогона модели. Переход к следующему прогону, если сделано менее N прогонов модели.

Строки 33—34. Вывод на печать среднего значения времени завершения работ, определенного по результатам N прогонов модели. Переход к исследованию сетевого графика со следующим значением параметра РЕСУРС, если  $РЕСУРС < РЕСМАХ$ .

Строка 35. Конец блока моделирования

Строка 36. Конец всей симула-программы.

В результате выполнения операторов в строках 25—35 примера 2.19 будут выданы средние значения времен завершения работ по сетевому графику для каждого исходного значения ресурсов в интервале от РЕСМИН до РЕСМАХ с шагом 1. В ходе моделирования на ЭВМ сетевого графика, изображенного на рис. 20, с помощью программы из примеров 2.17, 2.18, 2.19 были получены, например, следующие результаты:

## РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ

РЕСУРСЫ	СРОК ВЫПОЛНЕНИЯ
5	106.84
6	90.54
7	80.60
8	67.28
9	66.02
10	63.15
11	58.52
12	58.56

Рассмотренная программа моделирования может быть легко модифицирована так, что в результате каждого прогона будет определяться критический путь. Для этого достаточно в каждом процессе класса РАБОТА предусмотреть ссылочный

атрибут для указания узла, из которого исходит отображаемая им работа сетевого графика, и, используя этот атрибут, запоминать в процессах класса УЗЕЛ ссылку туда, откуда исходит входящая в данный узел работа, завершенная последней по времени. По окончании моделирования можно распечатать номера узлов, лежащих на критическом пути, пройдя по упомянутым ссылкам от завершающего узла сетевого графика к исходному узлу.

В декларацию класса УЗЕЛ (пример 2.17) для определения критического пути следует внести, например, на строку 2 описание атрибута ОППР **ref** (УЗЕЛ) ОППР, в который будет заноситься ссылка на узел, откуда пришла последняя работа.

Занесение требуемой ссылки в атрибут ОППР у объектов класса УЗЕЛ должна выполнять та работа, которая активизирует этот узел, поскольку именно она завершается позже всех других входящих в него работ. Для этого в декларации класса РАБОТА в совокупность формальных параметров (строка 1 пример 2.18) нужно внести атрибут УЗИСХ (узел исхода), специфицировав его как **ref** (УЗЕЛ) УЗИСХ, и в строке 11 вслед за символом **then** добавить оператор, который занесет в атрибут ОППР узла-цели ссылку на узел исхода запускающей его работы. Таким образом, строка 11 в примере 2.18 примет вид

```
if ЦЕЛЬ.КЗР=ЦЕЛЬ.ВХР then
begin ЦЕЛЬ.ОППР:=УЗИСХ;
  activate ЦЕЛЬ delay 0;
end;
```

В симула-программе примера 2.19 нужно предусмотреть еще один фактический параметр (У[А]) при генерации процессов класса РАБОТА (строка 23), соответствующий формальному параметру УЗИСХ.

Выдачу номеров узлов, лежащих на критическом пути, можно выполнить с помощью приведенного в примере 2.20 блока, который следует поместить после оператора **outfix** (TS/N, 2, 12) на строке 33 (пример 2.19).

Пример 2.20.

```
begin ref (УЗЕЛ) X;
  outtext ('КРИТИЧЕСКИЙ ПУТЬ:');
  for X:=У[НУЗЛ], X.ОППР while X/=none do
    outint (X.НОМЕР, 4);
end вывода критического пути;
```

В примере 2.20 предполагается, что последний элемент массива У указывает на завершающий узел сетевого графика. Отметим, что выводится критический путь, определенный

при последнем прогоне модели с данным количеством выделенных ресурсов, что делается во избежание загромождения распечатки. Целесообразно определять критический путь, задав детерминированные времена выполнения работ (все отклонения у работ положить равными 0) и число испытаний, равное 1.

2.6.3. Модель приоритетного обслуживания с прерываниями. В данном разделе мы рассмотрим применение средств моделирования языка симула-67 для имитации систем массового обслуживания, выполняющих обработку поступающих заявок в соответствии с их приоритетами. Будем считать, что система одновременно обслуживает одну заявку, а при поступлении заявки с более высоким приоритетом обработка текущей заявки прерывается и начинается обслуживание поступившей заявки, после окончания которого система возвращается к обслуживанию отложенной заявки. Ограничений на количество уровней прерывания накладывать не будем, т. е. обработка заявки, прерывавшей обслуживание заявки с меньшим приоритетом, может быть, в свою очередь, приостановлена в случае прибытия еще более приоритетной заявки. Заявки с одинаковыми приоритетами система обслуживает в порядке поступления. Время обслуживания каждой заявки будем считать равным значению одного из ее атрибутов.

Целью моделирования описанной системы выберем, например, определение коэффициента замедления обработки заявок в зависимости от их приоритета при заданном потоке заявок и распределении приоритетов. Коэффициент замедления будем вычислять как отношение разности времен начала и конца обработки данной заявки к времени обслуживания, заданному в соответствующем атрибуте. Если обслуживание заявки не прерывалось, то для нее коэффициент замедления будет равен 1.

Допустим, что поток заявок имеет интенсивность  $M$ , интервалы времени между поступлениями распределены по экспоненциальному закону, а приоритет каждой заявки с равной вероятностью принимает одно из целых значений  $1, 2, \dots, P$ . Время обслуживания заявок будем считать равномерно распределенным в интервале  $[A, B]$ . Следует заметить, что указанные допущения приняты только для определенности и не существенны для построения программы моделирования.

В примере 2.21 приведена декларация класса ГЕНЕРАТОР, описывающая процесс генерации заявок. В функции генератора входит создание объекта класса ЗАЯВКА с случайными значениями приоритета и времени обслуживания, передача этого объекта системе для обслуживания и повторение этих действий через случайные интервалы времени, распределенные по экспоненциальному закону с параметром  $M$ .



### Пример 2.21.

1. process class ГЕНЕРАТОР (N); integer N;
2. begin ref (ЗАЯВКА) X; integer I, U1, U2, U3;
3. U1:=1; U2:=3; U3:=5; I:=1;
4. ЗАПУСК: X:=new ЗАЯВКА (I, randint (1, P, U1),  
uniform (A, B, U2));
5. СМО. ОБРАБОТКА (X);
6. I:=I + 1; hold (negexp (M, U3));  
if I < N then goto ЗАПУСК;
7. end ГЕНЕРАТОР;

Поясним текст приведенной декларации класса.

Строка 1. Заголовок декларации класса ГЕНЕРАТОР. Фактическое значение атрибута N определяет максимальное число заявок, которое может быть создано.

Строка 2. Начало блока тела декларации. Описания ссылочной переменной X для обозначения созданного объекта класса ЗАЯВКА, целой переменной I для подсчета заявок и их нумерации, переменных U1, U2, U3, используемых в датчиках случайных чисел. Параметры законов распределения P, A, B, M будем считать глобальными величинами.

Строка 3. Присваивание начальных значений переменным.

Строка 4. Создание объекта X класса ЗАЯВКА с номером I и случайными значениями приоритета и времени обработки.

Строка 5. Передача заявки на обслуживание посредством обращения к процедуре ОБРАБОТКА, являющейся атрибутом объекта СМО. Глобальная переменная СМО указывает на процесс, отображающий систему массового обслуживания. В результате выполнения процедуры ОБРАБОТКА заявка X начнет обслуживаться или будет поставлена в очередь.

Строка 6. Увеличение номера заявки, задержка генератора на случайное время, распределенное по экспоненциальному закону, и переход на генерацию следующей заявки, если не достигнуто предельное число созданных заявок.

Строка 7. Конец декларации класса ГЕНЕРАТОР.

Объектом класса ЗАЯВКА никаких действий, кроме фиксации времени своего появления в атрибуте ТВХ, и присваивания начальных значений атрибутам ТНО и ТКО, мы приписывать не будем. В связи с этим соответствующая декларация класса, приведенная в примере 2.22, получается весьма простой и не нуждается в дополнительных пояснениях. Атрибуты ТНО и ТКО используются для фиксации времен начала и конца обслуживания данной заявки, а атрибут ТО, начальное значение которого равно полному времени обслуживания, задаваемому при генерации заявки в атрибуте ТОБСЛ, используется для

хранения остатка времени обслуживания в случае прерывания обработки данной заявки. Процедура B(S) вставляет заявку, атрибутом которой она является, в набор S, причем позиция заявки в этом наборе определяется ее приоритетом (атрибут ПРИОРИТЕТ): заявка встает в набор S вслед за всеми заявками с большим или равным приоритетом, но перед первой заявкой с меньшим приоритетом.

Пример 2.22.

```
link class ЗАЯВКА (НОМЕР, ПРИОРИТЕТ, ТОБСЛ);
  integer НОМЕР, ПРИОРИТЕТ; real ТОБСЛ;
begin real ТВХ, ТНО, ТКО, ТО;
  procedure B(S); ref (head) S;
  begin ref (ЗАЯВКА) X;
  comment вставляем заявку перед первым членом набора S с меньшим, чем у нее приоритетом. Если такого члена нет, то заявка ставится в конец S;
  X:—S.first;
  ПОИСК: if X==none then
  begin into (S); goto КОНЕЦ end;
  if X.ПРИОРИТЕТ < ПРИОРИТЕТ then
  ПЕРЕД X: begin precede (X);
  comment вставили перед X;
  goto КОНЕЦ; end;
  comment Заявки с одинаковыми приоритетами обрабатываются в порядке поступления;
  if X.ПРИОРИТЕТ = ПРИОРИТЕТ and X.ТВХ > ТВХ then
  goto ПЕРЕД X;
  X:—X.suc; goto ПОИСК;
  КОНЕЦ: end B;
  ТВХ: = time; ТНО: = -1; ТО: = ТОБСЛ;
end ЗАЯВКА;
```

З а м е ч а н и е. Отрицательное значение ТНО служит для системы обслуживания признаком того, что обработка данной заявки еще не начиналась.

Правила функционирования процесса класса СИСТЕМА, отображающего приоритетное обслуживание заявок, заданы в декларации, приведенной в примере 2.23.

Пример 2.23.

1. process class СИСТЕМА;
2. begin ref (head) ОЧЕРЕДЬ; ref (ЗАЯВКА) T;
3. procedure ОБРАБОТКА (X); ref (ЗАЯВКА) X;
4. begin if T==none then РАБОТА CX; begin T:—X;
5. X.ТНО:=time; reactivate СМО delay X.ТО end
6. else if X.ПРИОРИТЕТ ≤ T.ПРИОРИТЕТ then X.B(ОЧЕРЕДЬ)

```

7.     else ПЕРЕРЫВАНИЕ : begin T.TO:=СМО.evtime—time;
8.     T.B (ОЧЕРЕДЬ); goto РАБОТА CX end;
9.     end обработки поступившей заявки;
10.    ОЧЕРЕДЬ:—new head;
11.    РАБОТА: if ОЧЕРЕДЬ.empty then begin T:—none;
        passivate; goto КОВСЛ; end;
12.    T:—ОЧЕРЕДЬ.first; T.out; if T.HO<0 then T.THO:=time;
13.    hold (T.TO);
14.    КОВСЛ: T.TKO:=time; T.into (ОБРАБОТАННЫЕ);
15.    goto РАБОТА;
16.    end СИСТЕМА;

```

Дадим подробные пояснения к тексту примера 2.23.

Строки 1—2. Заголовок декларации класса СИСТЕМА и начало блока тела класса, в котором объявляются набор ОЧЕРЕДЬ и ссылочная переменная Т. В наборе ОЧЕРЕДЬ содержатся заявки, которые еще не начинали обслуживаться и те заявки, обслуживание которых было прервано. Заявки в этом наборе располагаются в порядке убывания приоритета, что обеспечивается процедурой В, определенной в классе ЗАЯВКА (см. пример 2.22). Переменная Т указывает на текущую заявку, обслуживаемую системой, либо равна none, если система простаивает.

Строка 3. Заголовок процедуры ОБРАБОТКА(X), которой в качестве параметра передается очередная заявка из входного потока. Эта процедура инициирует обслуживание поступающей заявки, если система бездействует или обслуживает заявку с меньшим приоритетом, а в противном случае отправляет поступившую заявку в очередь.

Строки 4—5. Если система простаивает, то начинается обработка поступившей заявки (X): текущей обрабатываемой заявкой становится X; в атрибуте ТНО заявки X отмечается время начала ее обслуживания; через интервал системного времени X.TO планируется возобновление работы процесса СМО, который выполнит действия, отображающие окончание обслуживания заявки. После выполнения строки 5 управление выходит из процедуры ОБРАБОТКА.

Строка 6. Эта строка выполняется в том случае, когда при поступлении новой заявки (X) система занята обслуживанием некоторой заявки Т. Если приоритет Т не меньше приоритета X, то заявка X включается в набор ОЧЕРЕДЬ на место, соответствующее ее приоритету, а система продолжает обслуживание заявки Т. В этом случае исполнение оператора X.B(ОЧЕРЕДЬ) заканчивает работу процедуры.

Строки 7—8. Эти строки выполняются тогда, когда приоритет прибывшей заявки X больше приоритета обрабатываемой за-

явки Т. При этом происходит прерывание обслуживания Т: ее атрибуту ТО присваивается интервал времени, которого системе не хватило для полной обработки заявки Т, сама заявка Т ставится в набор ОЧЕРЕДЬ в соответствии с ее приоритетом, а система переходит к обслуживанию заявки Х. Этот переход отображается выполнением операторов со строк 4—5, следующих за меткой РАБОТА СХ, которые переставляют указатель текущей заявки Т на заявку Х и перепланируют активную фазу процесса СМО на время, соответствующее окончанию обслуживания заявки Х. Локальное управление процесса СМО указывает в этот момент либо на оператор `goto` КОБСЛ (строка 11), если заявка Т застала систему в бездействии, либо на оператор с меткой КОБСЛ (строка 14), если в момент прихода Т система исполняла оператор `hold(T.TO)`, имитирующий обслуживание заявок, взятых ею из набора ОЧЕРЕДЬ. В обоих случаях процесс СМО во время своей очередной активной фазы выполнит строку 14, задающую действия по окончанию обслуживания новой текущей заявки, если, конечно, ее обработка не будет прервана приходом заявки с еще большим приоритетом.

Строка 9. Конец описания процедуры ОБРАБОТКА.

Строка 10. Начало операторной части тела класса СИСТЕМА. Заведение головы набора ОЧЕРЕДЬ, выполняемое во время первой активной фазы процесса СМО.

Строка 11. Если очередь заявок пуста, то система начинает простаивать. Простой отображается пассивным состоянием процесса СМО и равенством атрибута Т константе `none`.

Строка 12. Если в наборе ОЧЕРЕДЬ имеется хотя бы одна заявка, то система начинает обслуживать первую из них, поскольку, она обладает максимальным приоритетом. Первая заявка становится текущей, удаляется из очереди и в ее атрибуте ТНО фиксируется время начала обработки, если эта заявка ранее не начинала обслуживаться.

Строка 13. Имитация обработки заявки Т в течение времени Т.ТО. Если обслуживание Т будет прервано, то к моменту исполнения строки 14 процедура ОБРАБОТКА подменит значение Т, заставив эту переменную указывать на другую заявку, в то время как прерванная заявка помещается в набор ОЧЕРЕДЬ и находится там до тех пор, пока система снова не выберет ее оттуда при исполнении операторов строки 12. Таким образом, заявка может набирать требуемое ей время обслуживания за несколько фаз обработки.

Строка 14. Имитация конца обработки заявки. В атрибуте ТКО обработанной заявки фиксируется время окончания ее обслуживания и эта заявка заносится в набор ОБРАБОТАННЫЕ,

который используется при обработке и выдачи результатов моделирования.

Строка 15. Переход к обработке очередной заявки из очереди, которой может быть еще не обрабатывавшаяся заявка или заявка, вытесненная из системы более приоритетной заявкой.

Строка 16. Конец декларации класса СИСТЕМА.

Прежде чем писать оставшуюся часть программы моделирования приоритетного обслуживания, в которой иницируется работа модели и обрабатываются результаты имитационного эксперимента, рассмотрим, как проверить правильность функционирования модели. Анализировать работу имитационной модели удобно по протоколу, отражающему последовательность изменений ее состояния в системном времени. В данном случае наиболее наглядным протоколом моделирования обслуживания заявок была бы временная диаграмма работы модели с указанием номеров поступающих заявок, их приоритета, времени прихода, а также интервалов времени, в которые обслуживается каждая заявка. На рис. 21 изображена возможная форма

ВРЕМЯ	НОМЕРА ЗАЯВОН				Условные обозначения:
	1	2	3	4	
0 00	15 :				15 — Приоритет заявки. Печатается в момент появления в системе заявки.
1 00	*				
2 00	*				
3 00	*				
4 00	*				
5 00	* 20 :				
6 00	:	*			:
7 00	:	*			: — Интервалы ожидания обслуживания
8 00	:	*	10 :		:
9 00	:	*	:		
10 00	:	*	:		*
11 00	*		:		* — Фазы обслуживания заявки.
12 00	*		:		*
13 00	*		:		
14 00	*		:		
15 00	*		:		
16 00			*		
17 00			*		
18 00			*		

Рис. 21. Временная диаграмма работы модели приоритетного обслуживания.

временной диаграммы работы модели, которую можно получить в ходе имитационного эксперимента на печатающем устройстве ЭВМ.

Рассмотрим декларацию класса для процесса, который формирует и печатает временную диаграмму, приведенную на рис. 21, путем периодических наблюдений за состоянием модели,

### Пример 2.24.

```

1. process class ДИАГРАММА (T); real T;
2. begin boolean array БЫЛА[1:28]; ref (ЗАЯВКА) X;
   integer K;
3.   procedure ИНФ (X, C); ref (ЗАЯВКА) X; character C;
4.     begin integer I;
5.       if X==none then goto КОНЕЦ;
6.       I:=X.НОМЕР; sysout.image.setpos (15+4*I);
7.       outchar (C);
8.       if БЫЛА[I] then goto КОНЕЦ;
9.       sysout.image.setpos (13+4*I);
10.      outint (X.ПРИОРИТЕТ, 2);
11.      БЫЛА[I]:=true;
12.      КОНЕЦ; end ИНФ;
13. outtext (' ВРЕМЯ НОМЕРА ЗАЯВОК'); outimage;
14. outtext (blanks (15)); for K:=1 step 1 until 28 do
   outint (K, 4); outimage;
15. РАБОТА: outfix (time, 2, 14); outchar ("I");
16. for X:=СМО. ОЧЕРЕДЬ.first, X. suc while X/=none
   do ИНФ (X, ":");
17. ИНФ (СМО.T, "*"); outimage;
18. hold (T); goto РАБОТА;
19. end выдачи диаграммы;

```

### Пояснения.

Строка 1. Заголовок декларации класса ДИАГРАММА. Параметр T используется для задания периода, с которым процесс класса ДИАГРАММА выполняет обновление временной диаграммы работы модели.

Строка 2. Начало блока тела декларации класса ДИАГРАММА. Описание булевского массива БЫЛА длиной 28 элементов, используемого процедурой ИНФ при выдаче информации о состоянии заявок, находящихся в системе; объявление ссылочной переменной X для просмотра набора ОЧЕРЕДЬ и рабочей целой переменной K.

Строка 3. Заголовок процедуры ИНФ, которая обеспечивает занесение в буфер вывода (текстовая переменная image длиной 127 символов, определенная в объекте sysout и служащая образом строки печатающего устройства; более подробно см. главу 3) информации о заявке X. Местоположение выводимой информации определяется номером заявки. Параметр C используется для задания символа, отображающего состояние заявки. Процедура ИНФ помещает этот символ в крайнюю правую позицию четырехсимвольного поля, отводимого под информацию о заявке. Кроме того, если информация о заявке выводится пер-

вый раз, то в предыдущие две позиции этого поля заносится приоритет заявки. Первые 15 позиций каждой строки используются для печати моментов системного времени, в которые процесс класса ДИАГРАММА «снимает» состояние модели. Таким образом, информация о первой заявке располагается в позициях 16—19, о второй — в позициях 20—23 и т. д. Декларация класса в примере 2.24 рассчитана на выдачу информации о 28 первых заявках, обрабатываемых системой, чего вполне достаточно для отладки модели.

Строка 4. Начало блока тела процедуры ИНФ. Описание переменной I, используемой для хранения номера заявки X.

Строка 5. Если ссылка на заявку равна none, то никаких действий процедура не выполняет.

Строка 6. Занесение в переменную I номера заявки X. Установка указателя позиции в образе печатаемой строки на четвертый символ поля, предназначенного для информации о заявке X.

Строка 7. Занесение символа, отображающего состояние заявки X, в позицию строки, указанную предыдущим оператором.

Строка 8. Если I-й элемент массива БЫЛА равен true, то информация о приоритете заявки с номером I уже присутствует на временной диаграмме и выводить ее нет необходимости.

Строка 9. Установка указателя позиции в буфере вывода на второй символ поля для информации о заявке. Значение приоритета размещается в позициях 2—3 этого поля.

Строка 10. Размещение приоритета заявки X в соответствующих позициях буфера вывода.

Строка 11. Присваивание значения true I-му элементу массива БЫЛА. Следующий раз значение приоритета заявки с номером I выводиться не будет.

Строка 12. Конец описания процедуры ИНФ.

Строки 13—14. Выдача заголовка временной диаграммы. Оператор цикла обеспечивает нумерацию 4-х символьных полей, в которых будет располагаться информация о заявках: число 1, дополненное слева тремя пробелами, располагается в позициях 16—19, число 2 — в позициях 20—23 и т. д. Печать заголовка производится один раз при первой активной фазе процесса класса ДИАГРАММА.

Строка 15. Выдача значения системного времени в первые 14 позиций буфера вывода. В следующую, 15-ю позицию помещается символ I. Последовательность этих символов, напечатанных на различных строках временной диаграммы, но в одной и той же позиции, изображает ось времени.

Строка 16. Вывод информации о процессах, которые содержатся в наборе ОЧЕРЕДЬ и отображают заявки, обслуживание

которых было прервано или вообще не начиналось. Состояние этих заявок кодируется на временной диаграмме символом :.

Строка 17. Вывод на временную диаграмму информации о текущей обслуживаемой заявке, представляемой символом \*. Если в рассматриваемый момент времени система простаивает из-за отсутствия заявок, то в соответствующей строке временной диаграммы позиции с 16 по 127 будут заполнены пробелами. Оператор `outimage` выводит сформированный образ строки на печать и заполняет его пробелами.

Строка 18. Задержка процесса класса ДИАГРАММА на интервал системного времени, равный периодичности обновления временной диаграммы, и переход на формирование и выдачу следующей строки.

Строка 19. Конец декларации класса ДИАГРАММА.

Теперь мы имеем декларации классов, описывающие процессы генерации заявок (пример 2.21) и их обслуживания (пример 2.23), декларацию, задающую свойства самих заявок (пример 2.22), и, наконец, декларацию класса для процесса, делающего видимой работу модели (пример 2.24), позволяя тем самым эффективно вести семантическую отладку симула-программы. Нам остается лишь написать программное окружение для этих деклараций, в котором задать ввод исходных данных, запуск модели и обработку результатов моделирования. В примере 2.25 рассматривается симула-программа, с помощью которой удобно проводить отладку модели, задавая обработку небольшого числа (не более 20) заявок.

Пример 2.25.

1. **begin integer** P, K; **real** A, B, M, ТМОД, ТК;
2. P:=inint; K:=inint; M:=inreal; A:=inreal;
3. B:=inreal; ТМОД:=inreal; ТК:=inreal;
4. **outtext** ('МОДЕЛИРОВАНИЕ ПРИОРИТЕТНОГО ОБСЛУЖИВАНИЯ'); **outimage**;
5. **outtext** ('ДИАПАЗОН ПРИОРИТЕТОВ ОТ 1 ДО');  
**outint** (P, 3); **outimage**;
6. **outtext** ('МАКС. КОЛ-ВО ЗАЯВОК:'); **outint** (K, 4); **outimage**;
7. **outtext** ('ИНТЕНСИВНОСТЬ ПОТОКА:'); **outfix** (M, 2, 8);  
**outimage**;
8. **outtext** ('ВРЕМЯ ОБСЛУЖИВАНИЯ ОТ'); **outfix** (A, 2, 8);
9. **outtext** ('ДО'); **outfix** (B, 2, 8); **outimage**;
10. **outtext** ('ВРЕМЯ МОДЕЛИРОВАНИЯ:');  
**outfix** (ТМОД, 2, 12); **outimage**;
11. **outtext** ('ПЕРИОД КОНТРОЛЯ:'); **outfix** (ТК, 2, 8);  
**outimage**;
12. **simulation begin ref** (СИСТЕМА) СМО;  
**ref** (head) ОБРАБОТАННЫЕ; **ref** (ЗАЯВКА) У;



```

13. process class СИСТЕМА; ... (см. пример 2.23);
14. process class ЗАЯВКА; ... (см. пример 2.22);
15. process class ГЕНЕРАТОР; ... (см. пример 2.21);
16. process class ДИАГРАММА; ... (см. пример 2.24);
17. ОБРАБОТАННЫЕ:—new head;
18. СМО:—new СИСТЕМА; activate СМО;
19. activate new ГЕНЕРАТОР (K) delay 0;
20. activate new ДИАГРАММА (ТК) delay 0;
21. hold (ТМОД);
22. outtext ('РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ'); outimage;
23. outtext ('ЗАЯВКА ПРИОРИТЕТ ТОБСЛ ТВХ ТНО ТКО
ЗАМЕДЛЕНИЕ'); outimage;
24. if ОБРАБОТАННЫЕ.empty then begin
25. outtext ('НИ ОДНОЙ ЗАЯВКИ НЕ ОБРАБОТАНО');
goto КОНЕЦ end;
26. for Y:—ОБРАБОТАННЫЕ.first, Y.suc while
Y /= none do
27. begin outint (Y.НОМЕР, 5)); outint(Y. ПРИОРИТЕТ, 7);
outfix(Y.ТОБСЛ, 2, 9);
28. outfix (Y.ТВХ, 2, 9); outfix (Y.ТНО, 2, 9);
outfix (Y.ТКО, 2, 9);
29. outfix ((Y.ТКО—Y.ТВХ)/Y.ТОБСЛ, 2, 9); outimage;
30. end выдачи результатов;
31. КОНЕЦ: end блока simulation;
32. end симула-программы;

```

Пояснения к симула-программе примера 2.25.

Строка 1. Начало симула-программы. Объявление глобальных переменных:

Р — максимальное значение приоритета заявок;

К — предельное количество заявок, которые могут быть созданы во время работы модели;

А, В — нижний и верхний пределы возможных значений времени обслуживания заявок;

М — интенсивность потока заявок;

ТМОД — время моделирования (системное);

ТК — периодичность выдачи состояния модели на временную диаграмму.

Строки 2—3. Ввод начальных значений глобальных переменных. Данные для приведенных операторов могут быть подготовлены в виде следующей записи (перфокарты):

$P = 10$ ,  $K = 15$ ,  $M = 0.5$ ,  $ТОБСЛ = (0.5, 1.2)$ ,  $ТМОД = 30$ ,  $ТК = 0.25$

Строки 4—11. Распечатка введенных исходных данных. В результате работы приведенных операторов будет выведена следующая информация:

# МОДЕЛИРОВАНИЕ ПРИОРИТЕТНОГО ОБСЛУЖИВАНИЯ ДИАПАЗОН ПРИОРИТЕТОВ ОТ 1 ДО 10

МАКС. КОЛ-ВО ЗАЯВОК: 15

ИНТЕНСИВНОСТЬ ПОТОКА: 0.50

ВРЕМЯ ОБСЛУЖИВАНИЯ ОТ 0.50 ДО 1.20

ВРЕМЯ МОДЕЛИРОВАНИЯ: 30.00

ПЕРИОД КОНТРОЛЯ: 0.25

Строка 12. Начало блока моделирования. Объявление ссылочных переменных:

СМО — для обозначения процесса, отображающего систему обслуживания;

ОБРАБОТАННЫЕ — набор, куда заносятся полностью обслуженные заявки;

У — рабочая переменная для просмотра обработанных заявок при выдаче результатов моделирования.

Строки 13—16. Декларация классов СИСТЕМА, ЗАЯВКА, ГЕНЕРАТОР, ДИАГРАММА, взятые из примеров 2.21—2.24.

Строка 17. Создание набора ОБРАБОТАННЫЕ.

Строка 18. Создание процесса класса СИСТЕМА и исполнение его первой активной фазы, во время которой создается набор ОЧЕРЕДЬ.

Строка 19. Создание генератора заявок и планирование начала его работы на текущий момент времени.

Строка 20. Создание процесса, каждые ТК единиц времени выдающего информацию о состоянии модели в виде временной диаграммы, и планирование начала его работы на текущий момент времени.

Строка 21. Задержка дальнейших действий главной программы до момента системного времени ТМОД.

Строки 22—23. Выдача заголовка для результатов моделирования.

Строки 24—25. Если в наборе ОБРАБОТАННЫЕ не оказалось ни одной заявки, то выдается соответствующее сообщение и анализ результатов заканчивается.

Строка 26. Заголовок цикла для просмотра посредством переменной У всех членов набора ОБРАБОТАННЫЕ.

Строки 27—29. Выдача информации об одной обслуженной заявке.

Строка 30. Конец тела цикла, обеспечивающего выдачу информации обо всех обработанных заявках. После выполнения этого цикла будет напечатана следующая информация (при условии, что были обслужены 2 заявки).

## РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ

ЗАЯВКА ПРИОРИТЕТ ТОВСЛ ТВХ ТНО ТКО ЗАМЕДЛЕНИЕ

1	10	1.11	0.00	0.00	1.11	1.00
2	3	0.5	0.84	1.11	2.44	1.58

После завершения отладки приведенной в примере 2.25 симула-программы путем детального анализа ее работы при обслуживании небольшого количества заявок можно приступать к проведению имитационных экспериментов с целью получения статистических данных на основе большого числа испытаний. Однако предлагаемый в примерах 2.21—2.25 способ обработки результатов моделирования, при котором обработанные заявки записываются в набор и хранятся там до конца работы модели лишь для целей анализа содержащихся в них данных, не может применяться при достаточно большом числе заявок, обслуживаемых системой. Это связано с тем, что вместе с ростом числа обслуженных заявок пропорционально растет и занимаемая ими память, так как заявки являются членами доступного набора и, следовательно, не могут быть уничтожены.

Избежать указанного роста объема памяти, используемого при работе модели, можно при помощи перенесения части работы по сбору статистики на процесс, отображающий систему обслуживания. Его декларация класса приведена в примере 2.23. Например, если в результате моделирования нужно определить среднее значение замедления обработки заявок для каждого значения приоритета, то по окончании обслуживания заявки можно учесть ее замедление в элементе специального массива, соответствующем ее приоритету, а саму заявку — уничтожить. Напомним, что в языке симула-67 уничтожение объекта, т. е. объявление занимаемой им памяти свободной, производится автоматически в момент потери последней ссылки на этот объект.

Для учета количества заявок с определенным значением приоритета и получающихся значений замедления их обработки в блок `simulation` (пример 2.25, строка 12) вместо описаний набора ОБРАБОТАННЫЕ к переменной `У` следует поместить описания массивов `real array ЗАМЕДЛ [1:P]; integer array КЗ[1:P]` и описание целой рабочей переменной `J`. В `J`-м элементе массива `ЗАМЕДЛ` будет накапливаться сумма замедлений для заявок с приоритетом `J`, а в элементе `КЗ[J]` — общее количество обработанных заявок `J`-го приоритета.

В декларации класса СИСТЕМА (пример 2.23, строка 14) вместо оператора `T.into (ОБРАБОТАННЫЕ)` следует поставить операторы `J:=T.ПРИОРИТЕТ;`

$$\begin{aligned} \text{ЗАМЕДЛ}[J] &:= \text{ЗАМЕДЛ}[J] + (\text{T.ТКО} - \text{T.ТВХ}) / \text{T.ТОБСЛ}; \\ \text{КЗ}[J] &:= \text{КЗ}[J] + 1; \end{aligned}$$

которые заносят в массивы `ЗАМЕДЛ` и `КЗ` необходимую информацию об обработанной заявке `T`. Сама заявка `T` после этого уничтожается, поскольку при исполнении оператора `T:=none` (строка 11) или `T:=ОЧЕРЕДЬ.first` (строка 12) переменная `T`

принимает новое значение, а других ссылок на обработанную заявку в модели нет.

С введением указанных изменений несколько модифицируется и операторная часть блока simulation (пример 2.25):

— строку 17, в которой заводится голова набора ОБРАБОТАННЫЕ, следует опустить;

— вывод результатов моделирования (строки 23—30) будет состоять, по существу, из печати массивов ЗАМЕДЛ и КЗ, которую можно оформить в виде таблицы с помощью следующих операторов:

```
outtext ('ПРИОРИТЕТ КОЛ-ВО ЗАЯВОК СРЕДНЕЕ  
ЗАМЕДЛЕНИЕ'); outimage;  
for J:=1 step 1 until P do if КЗ[J] > 0 then  
begin outint (J, 6); outint (КЗ[J], 11);  
outfix (ЗАМЕДЛ[J]/КЗ[J], 2, 15); outimage;  
end вывода результатов;
```

Приведенные операторы отпечатают информацию о замедлении только для тех значений приоритетов, которые встречались хотя бы у одной обслуженной заявки.

В рассмотренной нами программе моделирования приоритетного обслуживания был предусмотрен один поток заявок со случайными временами обслуживания и приоритетами. Однако в программе легко можно задать несколько потоков заявок, каждый из которых имеет свое распределение времен обслуживания и приоритетов заявок. Для этого достаточно ввести атрибуты потока заявок в параметры генератора, т. е. заменить строку 1 в примере 2.21 на

```
process class ГЕНЕРАТОР (N, P, A, B, M); integer N, P;  
real A, B, M;
```

Значения глобальных переменных P, A, B, M, можно трактовать как некоторые базовые значения, используемые при задании фактических параметров для конкретных генераторов заявок. Например, если кроме основного потока из 1000 заявок с диапазоном приоритетов от 1 до P, временами обслуживания от A до B и интенсивностью M, задаваемого оператором

```
activate new ГЕНЕРАТОР (1000, P, A, B, M) delay 0;
```

нужно иметь еще один поток из 500 заявок с приоритетами от 1 до P/3, временами обслуживания от A/2 до 2\*B и интенсивностью M/10, первая заявка которого поступает в момент времени ТМОД/2, то для этого достаточно написать в главной программе оператор

**activate new** ГЕНЕРАТОР (500, P/3, A/2, 2\*B, M/10)  
**delay** ТМОД/2;

Для того чтобы отобразить поток заявок, в котором имеются определенные интервалы времени, когда появление заявок заведомо невозможно, следует при создании соответствующего процесса класса ГЕНЕРАТОР дать ему некоторое имя и переводить его в пассивное состояние оператором **cancel** в моменты приостановки. Возобновлять его работу можно с помощью операторов **activate** или **reactivate**.

Пусть, например, генератор ГЕН1 с интенсивностью 5, временами обслуживания от 1 до 5, приоритетами от 1 до 10 и общим количеством заявок 2000 должен иметь периоды неактивности в 5 единиц времени через каждые 15 единиц времени работы. Создание и первоначальный запуск этого генератора можно задать операторами

ГЕН1:—**new** ГЕНЕРАТОР (2000, 10, 1, 5, 5);  
**activate** ГЕН1;

Для периодического прерывания работы генератора ГЕН1 опишем специальный процесс, имеющий следующую декларацию класса:

**Пример 2.26.**

```
process class ПЕРЫВАТЕЛЬ;  
  begin M: hold (15); cancel (ГЕН1);  
        hold (5); activate ГЕН1; goto M  
  end ПЕРЫВАТЕЛЯ;
```

и запустим его в работу с помощью оператора

**activate new** ПЕРЫВАТЕЛЬ **delay** 0;

если первое прерывание должно быть через 15 единиц времени.

**2.6.4. Схема имитационной модели вычислительной системы.** В качестве примера имитационной модели вычислительной системы рассмотрим модель ЭВМ CRAY-1 [4]. В начале дается краткое описание архитектуры ЭВМ.

Высокая производительность CRAY-1 основывается на конвейерном принципе векторной и скалярной обработки информации и обеспечивается параллельной обработкой информации в нескольких конвейерах, а также путем организации зацепления конвейеров при выполнении векторных команд.

ЭВМ CRAY-1 осуществляет векторную и скалярную обработку 64-х разрядных слов с плавающей и фиксированной запятой, а также 24-х разрядных адресов. Обработка информации ведется с помощью 12-ти функциональных устройств. Все устрой-

ства условно можно разбить на 4 группы по типу выполняемых операций: адресные, скалярные, устройства с плавающей запятой и векторные. Каждое функциональное устройство реализует алгоритмы некоторого множества команд. Все устройства работают по конвейерному принципу и независимо друг от друга. Принцип конвейерной обработки команд основан на разбиении процесса выполнения операции на несколько последовательных этапов, каждый из которых выполняется в отдельном блоке устройства. Количество этапов обработки в конвейерном устройстве называется длиной конвейера (ДК). Операнды операции, выполняемой на конвейерном устройстве, вначале попадают на обработку в первый блок устройства. После обработки в первом блоке результаты передаются во второй, при этом первый блок освобождается и в него можно загрузить следующие операнды и т. д. Таким образом, количество команд, выполняемых одновременно на конвейерном устройстве равно ДК. Операция, начавшаяся в функциональном устройстве, завершается за фиксированное для заданного устройства время.

Функциональные устройства работают, по существу, в трех-адресном режиме: два регистра системы используются в качестве операндов, один регистр служит результатом.

**Регистры системы.** Всего имеется 8 адресных (A) регистров, 8 скалярных (S) регистров и 8 векторных (V) регистров по 64 компонента каждый. Функциональные устройства берут входную информацию из A, S и V регистров и выходную информацию размещают также в регистрах. Для увеличения производительности всей вычислительной системы на скалярных операциях введены буферные B и T регистры между памятью и соответственно A и S-регистрами. На рис. 22 представлена блок-схема регистров и функциональных устройств ЭВМ CRAY-1. Команда, которая в текущий такт выдается на исполнение, находится в регистре CIP, следующая за ней команда находится в регистре NIP. Выданная на исполнение команда должна удовлетворять определенным условиям. Эти условия различны для скалярных и векторных команд. Если условия выполнены, то команда начинает выполняться и завершает свои действия за фиксированное время, зависящее от того устройства, на котором она выполняется. Выполняемая команда может блокировать устройство и регистры, используемые в команде. Скалярная команда блокирует только регистр результата. Векторная команда блокирует векторные регистры, используемые в команде и устройство, на котором она выполняется. Блокирование распространяется на все время выполнения команды. Если условия не выполняются, то выполнение команды задерживается до тех пор, пока условия не будут выполнены.

Для примера рассмотрим как происходит выполнение векторной команды. Векторная команда посылает в первый такт работы в устройство первые компоненты операндов. Через такт

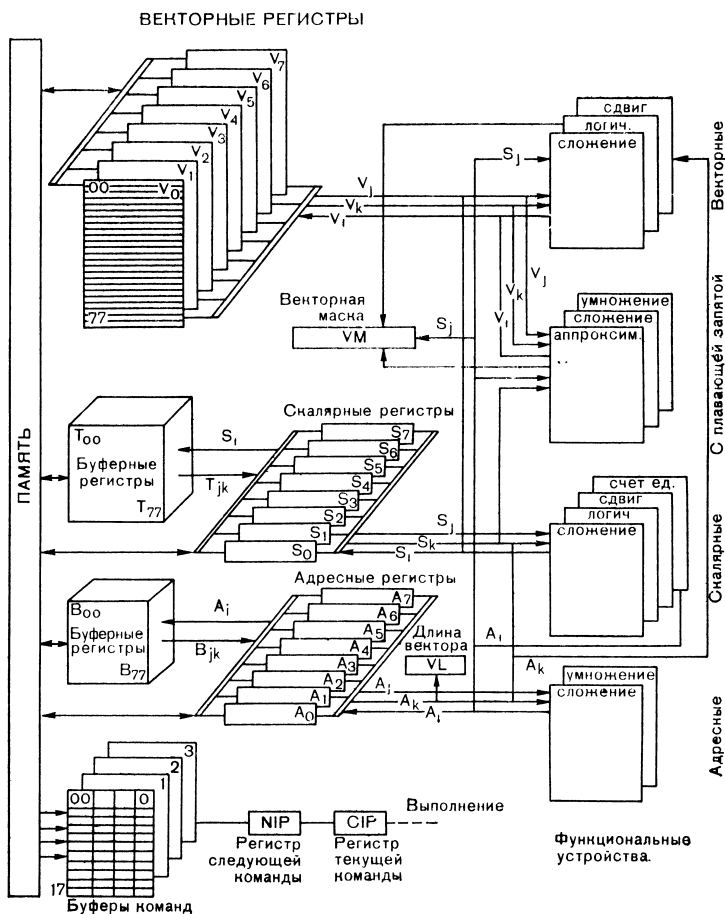


Рис. 22. Блок-схема регистров и устройств ЭВМ CRAY-1.

эти компоненты достигают устройства, а следующие компоненты готовы начать свое движение к устройству. Еще через такт будет наблюдаться следующая картина: первые компоненты векторов пройдут через первый блок устройства, вторые компоненты будут находиться перед входом в первый блок устройства, а третьи компоненты будут готовы начать свое движение к устройству. Через время, равное  $ДК + 2$ , первый компонент резуль-

тата будет находиться в регистре результата (еще один такт добавляется на передачу из устройства в регистр результата). После этого каждый такт будет выдаваться по одному компоненту результата. Например, время срабатывания устройства умножения с плавающей запятой равно 7 тактам. Если была задана команда умножения векторов  $V_2$  и  $V_3$  в виде  $V_1 = V_2 * V_3$ , то через 9 тактов будет получен первый компонент произведения:  $V_1[0] = V_2[0] * V_3[0]$ , а через  $K + 8$  тактов ( $K$  — длина векторов, участвующих в операции) будет полностью завершено покомпонентное перемножение векторов  $V_2$  и  $V_3$  с размещением результата в векторе  $V_1$ .

Если регистр-операнд векторной команды является заблокированным, то в общем случае это считается невыполнением условия для выдачи команды на исполнение. Однако если этот регистр является регистром результата в предыдущей команде (т. е. результат предыдущей команды используется в данной), то через время  $DK + 2$ , т. е. в момент получения первого компонента в регистре, можно начать команду (при условии, что все остальные условия на выдачу команды выполнены), в которой этот регистр является операндом. Рассмотрим последовательность двух команд, выполняющих покомпонентные операции над векторами:

$$V_1 := V_2 * V_3$$

$$V_4 := V_1 + V_5$$

Команда сложения использует в качестве операнда регистр  $V_1$ , являющийся регистром результата в команде умножения. Поэтому через 9 тактов после начала команды умножения начнет выполнение команда сложения, т. е. начиная с 9 такта работают одновременно две команды. Этот процесс носит название зацепления векторных команд.

**Имитационная модель CRAY-1.** Для проведения вычислительных экспериментов, связанных с оценкой производительности CRAY-1 на некоторых задачах, была разработана имитационная модель ЭВМ CRAY-1. На вход модели подается программа, записанная на языке, близком к автокоду ЭВМ GRAY-1. Модель осуществляет счет задач и сбор статистических данных, например, времен начала и окончания команд, времен и причин задержек команд перед выдачей на исполнение и т. п. Анализируя причины задержек, можно преобразовывать исходные программы для получения программ, более эффективных с точки зрения времени исполнения. В связи с тем, что модель реализована на языке высокого уровня, обладающем удобными средствами для моделирования, достаточно просто проводить модификацию модели, вводя новые архитектурные решения и оце-



нивая влияние этих изменений на время счета задач. Следует отметить ряд ограничений использования модели. Она позволяет проводить счет небольших программ (не более 400 команд входного языка). При получении временных характеристик на модели не учитывается время выборки команд из памяти. Считается, что все команды размещены в буфере команд.

Модель состоит из двух частей. В первой части задается описание функционирования устройств, регистров, команд системы и вспомогательные процессы и процедуры. Все программы этой части реализованы на языке симула-67. Во второй части представлены программы, интерпретирующие выполнение команд моделируемой системы. Программы второй части реализованы на автокоде.

Регистры, устройства, команды. Основными объектами модели являются объекты, соответствующие командам, регистрам и устройствам системы. Работа модели состоит в выполнении действий, задаваемых в командах. Объекты, соответствующие регистрам и устройствам системы, представляют собой структуры данных, на которые воздействуют команды.

Описание каждого типа регистра задается декларацией соответствующего класса. Для ссылок на объекты этих классов используются ссылочные переменные. Таким образом, каждый регистр моделируется объектом, а для ссылки на него используется ссылочная переменная. Например, описание векторного регистра имеет следующий вид:

```
РЕГИСТР class REGV;  
  begin boolean ЗАНЯТ; integer ВРЕЗАН, ТУСТ;  
    array ЗНАЧ [0 : 127];  
  end;
```

Переменные ЗАНЯТ, ВРЕЗАН, ТУСТ используются для организации зацепления векторных команд. В массиве ЗНАЧ хранятся 64 компонента вектора. В связи с тем, что слово на CRAY состоит из 64-х разрядов, а слово на БЭСМ-6 из 48-ми разрядов, то под каждое слово CRAY отводится 2 слова БЭСМ-6. Кроме того, в классе РЕГИСТР определяются некоторые общие для всех регистров атрибуты. Для ссылок на векторные регистры используется ссылочный массив

```
ref (REGV) array V [0 : 7];
```

Генерация объектов, моделирующих векторные регистры, выполняется оператором

```
for I:=0 step 1 until 7 do  
  V [I]:—new REGV;
```

Устройства системы также рассматриваются как структуры данных и их определение и генерация выполняются аналогично.

В качестве процессов мы определяем команды системы. Структура команды представлена на рис. 23.

```
process class КОМАНДА (КОД, РЕЗ, ОП1, ОП2);  
  ref (РЕГИСТР) РЕЗ, ОП1, ОП2;  
  integer КОД;  
begin НАЧАЛО: ПРОВЕРКА УСЛОВИЙ НА ВЫДАЧУ КОМАНДЫ;  
  УСТАНОВКА СТАТИСТИЧЕСКИХ ПЕРЕМЕННЫХ;  
  БЛОКИРОВКА РЕГИСТРОВ И УСТРОЙСТВА (ЕСЛИ ЭТО  
  НЕОБХОДИМО);  
  ВЫПОЛНЕНИЕ КОМАНДЫ;  
  ОСВОБОЖДЕНИЕ РЕГИСТРОВ И УСТРОЙСТВА;  
  ЗАВЕРШЕНИЕ КОМАНДЫ;  
end;
```

Рис. 23. Схема класса КОМАНДА.

Здесь приведена упрощенная схема команды. Предполагается, например, что в команде в качестве операндов и результата выступают регистры. В некоторых командах в качестве квалификации регистров может выступать конкретный вид регистра (векторный, скалярный и т. п.). С другой стороны, существуют команды, в которых в качестве операнда выступает числовое значение. В модели существуют декларации классов для различных типов команд. Каждой декларации соответствует набор команд, которые выполняются на конкретном устройстве. Числовой код операции задается значением переменной КОД.

Рассмотрим, какие действия выполняет процесс класса КОМАНДА. В начале работы необходимо провести проверку условий на выдачу команды. Если условия не выполнены, то команда переходит в пассивное состояние (выполняется оператор *passivate*). В дальнейшем через такт работы системы процесс ТОЛКАЧ проведет запуск команды и она повторит проверку условий. Так будет продолжаться до тех пор, пока условия не будут выполнены и команда перейдет к исполнению своих действий. Если условия выполнены, то производится обновление значений переменных, используемых для сбора статистики о времени и причинах задержки выполнения команды. Некоторые команды вызывают блокировку регистров операндов и функциональных устройств и поэтому в модели необходимо отображать эти блокировки. В связи с тем, что время блокировки обычно не совпадает с временем исполнения самой команды, для этой цели используется вспомогательный процесс. Задача этого процесса состоит в том, чтобы установить соответствующие при-

знаки блокировки в указанных регистрах и спустя определенное время сбросить эти признаки. В векторных командах необходимо в регистр результата установить специальные признаки для последующей организации зацепления (если, конечно, оно возможно). После выполнения этих действий проводится интерпретация выполнения самой команды и задержка процесса КОМАНДА на время, необходимое для выполнения команды. В заключение, если это необходимо, разблокируются использованные регистры и команда пассивируется. При последующей активизации команды (т. е. при необходимости повторного выполнения этой команды) выполняется переход из конца тела команды на метку НАЧАЛО и команда повторяет все свои действия.

Декларация класса КОМАНДА отображает общие свойства, присущие всем типам команд. Имея такую декларацию, при описании конкретных команд достаточно отобразить лишь их специфические особенности, сославшись на описание общих свойств при помощи префикса КОМАНДА. В программе модели имеется, таким образом, целая иерархия деклараций классов, описывающих команды моделируемой ЭВМ.

Ниже, в качестве примера приведена декларация класса, описывающая алгоритм команды сложения векторов.

```
КОМАНДА class KOMAVC (PE3, ОП1, ОП2);
  ref (REGV) PE3, ОП2; ref (РЕГИСТР) ОП1;
1.  begin ВЕКТВЫДКОМ (AVC, PE3, ОП1, ОП2);
      comment AVC — ссылка на устройство;
2.    AVC.СВОБОДЕН:=false;
3.    НАЧКОМ (НОМ);
4.    VECTOR (PE3);
      comment блокировка регистров команды;
5.    activate new БЛОК (PE3, AVC.T + 1) after current;
      if ОП1 is REGV then
6.      activate new БЛОК (ОП1, I — 2) after current;
7.      activate new БЛОК (ОП2, I — 2) after current;
      comment интерпретация выполнения команды;
8.    СЧЕТАV (КОП, PE3, ОП1, ОП2);
      comment задержка на время исполнения;
9.    hold (I + 2);
      comment освобождение устройства;
10.   AVC.СВОБОДЕН:=true;
11.   КОНКОМ (НОМ);
      end KOMAVC;
```

Декларация имеет в качестве префикса класс КОМАНДА. В этом классе задаются атрибуты и общие действия для всех

команд системы. Параметрами класса КОМАНД являются три ссылочные переменные, каждая из которых ссылается на определенный регистр. Данная команда является векторной и регистр результата (РЕЗ) и один операнд (ОП2) в этой команде будут векторными, а один операнд (ОП1) может быть либо векторным регистром, либо скалярным. В связи с этим параметр ОП1 квалифицирован классом РЕГИСТР, который является префиксом к декларациям классов, описывающих конкретные регистры.

Строка 1. Вызывается процедура ВЕКТВЫДКОМ. В этой процедуре осуществляется проверка условий на выдачу команды. В дополнение к регистрам, в этой процедуре в качестве параметра используется ссылка на векторное устройство сложения. Если условия для выдачи команды на исполнение не выполнены, то процедура приостанавливает свое выполнение с помощью оператора *passivate*. Ее следующая активизация произойдет из процесса ТОЛКАЧ, который каждый такт активизирует команды. Если условия выполнены, то процедура завершает свои действия.

Строка 2. Блокируется устройство.

Строка 3. Процедура НАЧКОМ производит установку некоторых переменных для сбора статистических данных.

Строка 4. Процедура VECTOP проводит установку значений переменным в векторном регистре, с помощью которых организуется зацепление векторных команд.

Строки 5—7. Генерируются процессы класса БЛОК. Данный процесс предназначен для блокировки векторных регистров на определенное время. Первый параметр указывает регистр, который надо блокировать, второй — время, на которое проводится блокировка. Регистр ОП1 может быть либо скалярным, либо векторным. Его блокировка необходима лишь в том случае, если он векторный. Поэтому предварительно осуществляется проверка, является ли операнд ОП1 векторным регистром.

Строка 8. Вызывается процедура СЧЕТАУ. Эта процедура предназначена для интерпретации выполнения операции.

Строка 9. Задерживается выполнение процесса на время исполнения команды. Это время зависит от длины вектора и вычисляется в классе КОМАНДА.

Строки 10—11. После того как завершено выполнение команды, освобождается устройство и проводится установка переменных для сбора статистики.

Если в программе, которая должна считаться на модели, встретится команда векторного сложения, например:

$$V1 := V2 + V3,$$

то генерация этой команды выполняется с помощью

**new KOMAVC (КОД, V[1], V[2], V[3])**

где КОД — код операции, V[1], V[2], V[3] — ссылки соответственно на первый, второй, третий векторные регистры.

**Моделирование памяти.** Объем оперативной памяти CRAY-1 составляет  $10^6$  слов. Моделирование такой памяти на инструментальной ЭВМ затруднительно. Кроме того, для целей, которые ставятся перед моделью, этого и не требуется. Естественно, под память отвести одномерный массив некоторой длины. Следует, однако, отметить, что длина массива не может быть большой, так как в поле данных (оперативная память, остающаяся после загрузки программ модели) должны размещаться объекты, отображающие структуру ЭВМ и объекты, соответствующие командам. Желание проводить на модели счет как можно больших по размеру программ вынуждает ограничивать длину массива, отводимого под память. Под память моделируемой системы отводится массив, состоящий из 1024 слов. Каждые два элемента этого массива моделируют одно слово системы. Таким образом, элементы этого массива моделируют 512 слов памяти вычислительной системы. Вся моделируемая память разбивается на страницы. В оперативной памяти находится одна страница и ее номер содержится в переменной НОМЕР ЛИСТА. Оставшиеся страницы памяти хранятся во внешней памяти. Такой подход позволяет существенно увеличить размер моделируемой памяти. При обращении к памяти (чтение или запись) анализируется адрес слова, используемого в команде. Если слово не принадлежит текущему листу памяти, т. е. тому листу, который находится в настоящее время в массиве, то необходимо провести подкачку требуемого листа. Массив ПАМЯТЬ описывается в программе следующим образом:

**common ПАМЯТЬ (1024); array ПАМЯТЬ [0 : 1023];**

**З а м е ч а н и е.** Описатель **common**, реализованный в симула-компиляторах для БЭСМ-6 и ЕС ЭВМ (см. 4.2) делает массив ПАМЯТЬ доступным для программ обмена с внешней памятью, которые из соображения эффективности написаны в данной модели на автокоде. При использовании для организации обменов средств языка симула-67 употребление описателя **common** становится ненужным.

**В в о д п р о г р а м м ы.** Алгоритм, который необходимо просчитать на модели, записывается на языке, близком к автокоду CRAY-1. Каждое предложение программы набивается на отдельной перфокарте. В связи с тем, что каждая команда представляется процессом, необходимо по каждому предложению

программы сгенерировать объект соответствующего класса. В модели имеется процедура, загружаемая динамически, с помощью которой из текстового представления команды генерируется объект класса КОМАНДА. В процедуре широко используются стандартные средства по работе с текстами. Полученные объекты ставятся в набор ОЧЕРЕДЬ, моделирующий буфер команд. Кроме задачи генерации объектов для команд заданных типов, в процедуре осуществляется привязка меток из команд перехода к меткам, стоящим перед командами.

Суть алгоритма привязки состоит в следующем. При анализе команды определяется наличие метки в соответствующем поле команды. Если в команде указана метка, то создается уведомление об этой метке. Уведомление содержит в себе ссылку на данную команду и текстовое значение самой метки. Уведомление представляется в модели в следующем виде:

```
link class ИОМ (КОМ, МЕТКА);  
value МЕТКА;  
ref (КОМАНДА) КОМ; text МЕТКА;;
```

Созданное уведомление ставится в набор. Описание и генерация набора производится с помощью описателя

```
ref (head) ОПРЕДМЕТКИ;
```

и оператора

```
ОПРЕДМЕТКИ :— new head;
```

Теперь, если предположить, что ссылка на команду, перед которой стоит метка, хранится в ссылочной переменной КОМ, а ссылка на текст метки в текстовой переменной ТМЕТКИ, то постановка уведомления в набор выполняется оператором

```
new ИОМ (КОМ, ТМЕТКИ).into (ОПРЕДМЕТКИ);
```

Если в программе встретилась команда перехода, то уведомление об этой команде ставится в набор МЕТКИНЕОПР, где содержатся все уведомления о командах перехода.

Перед началом исполнения программы на модели необходимо разрешить метки, т. е. связать метки в командах перехода с соответствующими метками перед командами. В нашем случае в атрибут объекта класса команды перехода необходимо поставить ссылку на объект команды, в которой эта метка определена.

В объекте класса КОМАНДЫ ПЕРЕХОДА атрибут, который ссылается на следующую исполняемую команду, имеет следующее описание:

```
ref (КОМАНДА) КУДА;
```

Ниже приведен фрагмент программы модели, реализующий разрешение меток.

```
1. if МЕТКИНЕОПР.cardinal≠0 then
2.   for K:—МЕТКИНЕОПР.first while K≠/=none do
3.     begin if ОПРЕДМЕТКИ.cardinal=0 then ОШИБКА else
4.       for K1:—ОПРЕДМЕТКИ.first, K1.suc while K1≠/=none do
5.         if K. ТМЕТКИ=K1. ТМЕТКИ then
6.           begin K. КОМ. КУДА:—K1. КОМ; goto ВЫХОД; end;
7.         ОШИБКА;
8.       ВЫХОД: K. out;
   end;
```

Рассмотрим подробнее операторы данного фрагмента.

Строка 1. В начале проводится проверка набора МЕТКИНЕОПР. Если набор пуст, т. е. в программе нет команд перехода, то приведенные операторы не выполняются. Если набор не пуст, то переходим к выполнению фрагмента.

Строка 2. В цикле выбираем первое уведомление о команде перехода.

Строки 3—4. Если набор, содержащий уведомления о помеченных командах, пуст, то выполняется процедура ОШИБКА, т. е. в программе есть команды перехода, а меток нет. В противном случае, последовательно просматриваем все уведомления из набора ОПРЕДМЕТКИ.

Строки 5—6. Если метка из уведомления о команде перехода совпадает с меткой из команды, где она определена, то в объект класса КОМАНДА ПЕРЕХОДА в атрибут КУДА надо занести ссылку на команду, где эта метка определена, и перейти к метке ВЫХОД.

Строка 7. Если метка из команды перехода не совпала ни с одной меткой из набора ОПРЕДМЕТКИ, то выполняется процедура ОШИБКА.

Строка 8. Уведомление о команде перехода исключается из набора.

Функционирование модели. В начале моделирования необходимо произвести заполнение памяти и регистров модели начальными данными для счета конкретной задачи. После того, как память заполнена, производится ввод текста программы, которую мы хотим считать на модели. После завершения ввода программы все команды оказываются размещенными в буфере команд. Теперь можно приступить к выполнению программы. Старт программы задается с помощью активизации процесса ТОЛКАЧ. Приведем декларацию класса ТОЛКАЧ.

```
process class ТОЛКАЧ;
  begin СІР. ЗНАЧ:—ОЧЕРЕДЬ. first;
```

```

NIP.ЗНАЧ:—CIP.ЗНАЧ.suc;
L1: if CIP.ЗНАЧ==none then goto ВЫХОД;
    reactivate CIP.ЗНАЧ after current;
    hold (1); СМЕНА;
    ВЫХОД;
end ТОЛКАЧ;

```

В начале работы данного процесса производится присваивание значениям регистров CIP и NIP соответственно ссылок на первую и вторую команды из очереди. Далее идет циклическая часть программы. В начале проверяется значение регистра текущей команды. Если это значение равно **none**, т. е. выполнялась последняя команда, то процесс завершает свою работу. Если значение регистра отлично от **none**, то активизируется та команда, на которую указывает значение регистра CIP и процесс задерживается на 1 такт. При его следующей активизации сначала выполняется процедура СМЕНА. Данная процедура изменяет переменные ЗНАЧ в регистрах CIP и NIP. Это изменение можно выполнить следующими операторами:

```

CIP.ЗНАЧ:—NIP.ЗНАЧ;
NIP.ЗНАЧ:—NIP.ЗНАЧ.suc;

```

**З а м е ч а н и е.** В модели перед изменением значений проводится ряд проверок условий. Во-первых, следует учитывать, что текущая команда может быть командой перехода и поэтому регистр CIP, например, должен принимать значение в зависимости от условия перехода в команде. Во-вторых, если в текущей команде не выполнены условия на выдачу команды, то значения регистров не изменяются.

После того как вся программа завершит свои действия, работа модели переходит в заключительную стадию, связанную с обработкой полученных результатов. В частности, можно получить распечатку состояния значений памяти, всех регистров системы, времена задержек команд перед выдачей на исполнение и т. д.



## ГЛАВА 3

### ОБРАБОТКА ТЕКСТОВ

Кроме традиционных возможностей по обработке чисел в язык симула-67 введены стандартизованные средства по работе с литерами и текстами. Разработанный авторами языка аппарат работы с текстами позволяет осуществлять различные преобразования над текстами: замена части текста на другой текст, преобразование числовых значений в текстовые и наоборот и т. п. Эти средства представляют широкие возможности по редактированию вводимых и выводимых данных, созданию и обработке файлов, размещенных на внешних устройствах ЭВМ.

#### 3.1. Литеры

Для работы с отдельными символами в языке симула-67 предусмотрен специальный тип значений — литеры. Примерами литерных значений служат буквы алфавита, цифры, специальные знаки типа +, \*, (,) и т. п. Литерные значения (константы) заключаются в две кавычки. Например, "А", "+" и т. д. Значения типа литеры могут присваиваться переменным такого же типа (простым и с индексами), передаваться в качестве параметров в процедуры и объекты, вычисляться в результате выполнения процедуры-функции.

Литерные переменные, а также массивы и процедуры-функции, используемые в программе, должны быть описаны с помощью описателя **character**. Например:

```
character X, Y, Z;
```

```
character array АЛФАВИТ [1 : 33];
```

```
character procedure ЗАМЕНА (X); character X; ...;
```

Присваивание значений литерным переменным осуществляется оператором присваивания (**:=**). Значение, полученное в результате вычисления правой части, присваивается переменной, стоящей в левой части. Например,

**character** C1, C2;

C1:="A",

C2:=C1;

В результате выполнения этих операторов значения переменных C1 и C2 будут равны "A".

**3.1.1. Упорядоченность значений литер.** Отношения литер. Количество литер, которые можно использовать в программе, зависит от конкретных вычислительных систем и при переходе с одной системы на другую может изменяться. Существенно может отличаться количество литер на устройстве подготовки данных от количества литер на АЦПУ. В результате, диапазон изменения значений для литерных переменных при каждой реализации может различаться. Таблицы допустимых литерных значений для трансляторов на БЭСМ-6 и ЕС ЭВМ приведены в Приложении 6.

В языке симула-67 предусмотрена упорядоченность множества литерных значений: каждой литере ставится во взаимно однозначное соответствие целое положительное число, называемое рангом литеры. Преобразование литер в целые числа и обратно выполняется с помощью двух функций:

**integer procedure** rank (C); **character** C; ...;

**character procedure** char (K); **integer** K; ...;

Первая процедура доставляет в качестве своего результата ранг фактического параметра. Вторая процедура по заданному рангу K доставляет литерное значение, соответствующее K. Диапазон значений, как уже отмечалось, зависит от конкретных реализаций, и поэтому при использовании процедуры char следует учитывать тот факт, что не для всякого K найдется конкретное литерное значение. Если мы обратимся к процедуре с некоторым K, для которого нет соответствующей литеры, то в результате будет выдано сообщение об ошибке.

В реализациях языка симула-67 на ЭВМ БЭСМ-6 и ЕС ЭВМ ранг литеры вычисляется как числовое значение ее представления в коде ISO [18] (для БЭСМ-6) или в коде ДКОИ [13] (для ЕС ЭВМ).

Введенное соответствие между литерами и числами позволяет определить операции сравнения литер. Для записи операции отношения можно использовать знаки  $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $\geq$ ,  $>$ , которые имеют обычный смысл. Отношение двух литерных значений

X оп Y,

где оп — некоторая операция отношения, имеет логическое

значение, совпадающее со значением отношения их рангов

rank (X) оп rank (Y).

**3.1.2. Подмножества литер.** Все множество литер можно разбить на три подмножества: буквы, цифры, прочие знаки. В языке имеются две процедуры, с помощью которых определяется принадлежность литер к заданному подмножеству:

**boolean procedure digit (C); character C; ...;**  
**boolean procedure letter (C); character C; ...;**

Значением процедуры digit будет **true**, если литера (фактический параметр) является цифрой, и **false** — в противном случае. Значением процедуры letter будет **true**, если литера (фактический параметр) является буквой.

**Пример 3.1.** Здесь приводится программа, позволяющая установить, какие числовые значения соответствуют символам устройства подготовки данных. На вход в программу подается набор символов, набитых на перфокарте. В программе выделяются буквы, цифры и прочие знаки и распечатываются их ранги, полученные с помощью функции rank.

```
begin integer K, K1, K2, K3; character array C1 [1:3, 1:50];  
  K1:=K2:=K3:=1;  
  comment в первой строке матрицы C1 будут размещены  
  буквы, во второй — цифры, в третьей — прочие знаки;  
  while not lastitem do  
    begin C := inchar;  
    if letter (C) then  
      begin C1 [1, K1] := C; K1 := K1 + 1; goto L end;  
    if digit (C) then  
      begin C1 [2, K2] := C; K2 := K2 + 1; goto L end;  
    C1 [3, K3] := C; K3 := K3 + 1; L:  
    end;  
  comment ниже приводятся операторы для печати в три  
  столбца литер (первый столбец — буквы, второй — цифры,  
  третий — прочие знаки) и их числовых значений;  
  for K:=1 step 1 until K1 - 1 do  
    begin outchar (C1 [1, K]); outtext (' : ');  
    outint (rank (C1 [1, K]), 3); outtext (' ');  
    if K ≤ K2 - 1 then  
      begin outchar (C1 [2, K]); outtext (' : ');  
      outint (rank (C1 [2, K]), 3); outtext (' ');  
      end;  
    if K ≤ K3 - 1 then  
      begin outchar (C1 [3, K]); outtext (' : ');
```

```

        outint (rank (C1 [3, K]), 3);
    end;
    outimage;
end;
for K := 21 step 1 until 126 do
    begin outchar (char (K)); outtext (' : ');
        outint (K, 3); outimage;
    end;
end

```

**З а м е ч а н и е.** В программе используются операторы ввода-вывода языка симула-67 (см. 3.3). Здесь укажем, что функция `inchar` осуществляет ввод литеры с перфокарты и в качестве своего значения доставляет очередную литеру из входного потока. Процедуры `outchar`, `outtext` и `outint` предназначены для вывода на печать соответственно литеры, текста и значения целого. Процедура `outimage` выполняет перевод строки АЦПУ.

## 3.2. Тексты

В языке симула-67 имеется возможность обработки текстов, представляемых упорядоченными последовательностями литер. На понятии текста базируются средства ввода-вывода. Значения текстов, представленные явным образом в программе, называются текстовыми константами и заключаются в апострофы ('), например: 'СИМУЛА-67'. Количество литер в тексте называется *длиной текста*. В общем случае удобно считать, что текстовое значение заключено в некоторый объект, который в дальнейшем будем называть текстовым объектом. Для того чтобы работать с текстами, необходимо дать им имена. Для этих целей используются переменные типа `text`. Определение текстовых переменных делается обычным способом. Например:

```

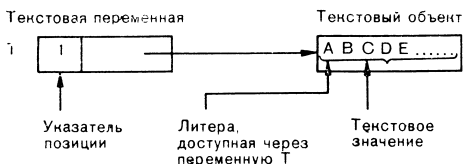
text T1, T2;
text procedure strip; ...;
text array ТАБЛИЦА [1 : N];

```

Начальное значение текстовых переменных равно `notext`, т. е. описанные переменные ссылаются на пустой текст. Пустой текст — это текст, не содержащий ни одной литеры.

Организация работы с текстами во многом напоминает объекты и действия над ссылочными переменными. Текстовый объект содержит атрибуты-функции, которые позволяют проводить гибкую работу с текстовым значением. Обращение к атрибутам текстового объекта осуществляется при помощи дистанционного доступа, где слева от точки указывается ссылка на текстовый объект. Ниже будут рассмотрены процедуры, осу-

ществляющие работу с отдельными литерами текста. Литеры текстового значения доступны по одной. В ссылке на текст содержится указатель позиции, который ссылается на литеру текста, доступную в данный момент. Таким образом, текстовая переменная содержит в себе ссылку на текстовый объект и указатель позиции на доступную литеру в текстовом значении этого объекта. Ниже приведена логическая структура текстовой переменной и текстового объекта.



**3.2.1. Создание текстовых объектов. Подтексты.** Создание новых текстовых объектов осуществляется встроенными процедурами:

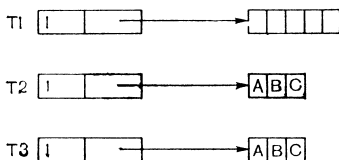
```
text procedure blanks (N); integer N; ...;  
text procedure text (T); value T; text T; ...;
```

Процедура **blanks** доставляет в качестве значения ссылку на вновь сгенерированный текстовый объект. Текстовое значение заполняется *N* литерами пробела. Указатель позиции получает значение 1. Процедура **text** также доставляет в качестве значения ссылку на новый текстовый объект. Если в качестве фактического параметра используется текстовая ссылка, то новый объект является копией объекта, на который ссылается фактический параметр. Если в качестве параметра используется текстовая константа, то текстовое значение нового объекта заполняется литерами этой константы. Значение указателя позиции равно 1, а длина текстового значения равняется количеству литер текстовой константы.

**Пример 3.2.**

```
text T1, T2, T3;  
T1:— blanks (5);  
T2:— text ('ABC');  
T3:— text (T2);
```

После выполнения этих операторов возникает следующая логическая структура данных:



Для организации работы с частью текстового значения в языке имеются процедуры

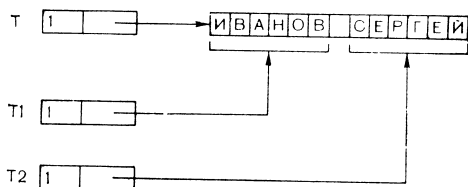
```
text procedure sub (K, J); integer K, J; ...;  
text procedure strip; ....;
```

Процедура  $T.sub(K, J)$  в качестве своего значения доставляет ссылку на подтекст в тексте  $T$ , начиная с  $K$ -й литеры длиной в  $J$  литер. Необходимо учитывать, что значения параметров  $K$  и  $J$  не должны выходить за пределы текста  $T$ . Пусть  $N$  длина текста  $T$ . Тогда обращение  $T.sub(K, J)$  допустимо, если  $K \geq 1$  и  $K + J \leq N$ . Если  $K$  и  $J$  не удовлетворяют этим условиям, то при выполнении процедуры выдается сообщение об ошибке. Значение указателя позиции в ссылке равно 1, т. е. указатель позиции указывает на первую литеру в выделенном подтексте ( $K$ -ю литеру в тексте  $T$ ). Указатель позиции для переменной  $T$  остается без изменения. Результатом выполнения  $T.sub(K, 0)$  является **notext**.

Пример 3.3. Пусть заданы следующие описания и операторы:

```
text T, T1, T2;  
T:— text ('ИВАНОВ СЕРГЕЙ');  
T1:— T.sub(1, 6);  
T2:— T.sub(8, 6);
```

В результате выполнения этих операторов будет построена следующая логическая структура:

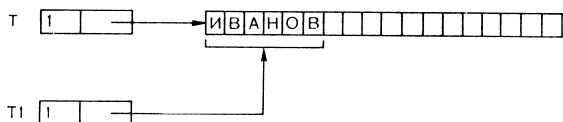


Рассмотрим вторую процедуру работы с подтекстами — **strip**. Обращение  $T.strip$  эквивалентно обращению  $T.sub(1, N)$ , где  $N$  такое число, что литеры текстового значения  $T$  в позициях под номерами больше  $N$  все суть литеры пробела. Если текстовое значение  $T$  состоит только из литер пробела, то выражение  $T.strip$  будет иметь значение **notext**.

Пример 3.4. Пусть заданы следующие описания и операторы:

```
text T, T1;  
T:— text ('ИВАНОВ _____');  
T1:— T.strip;
```

После выполнения этих операторов будет построена следующая структура:



3.2.2. Присваивание текстовых ссылок и текстовых значений. Процедуры `blanks` и `text` в качестве значения доставляют ссылку на новый текстовый объект. Для того чтобы запомнить эту ссылку, необходимо использовать оператор присваивания ссылок. Неформально присваивание текстовых ссылок можно представить в следующем виде:

$T:-X$

где  $X$  — выражение, значением которого является текстовая ссылка,  $T$  — текстовая переменная (простая или с индексами).

В результате выполнения оператора присваивания текстовых ссылок переменная  $T$  будет ссылаться на тот же текст, что и  $X$ . Кроме этого, в переменную левой части копируется и указатель позиции из ссылки, заданной в правой части.

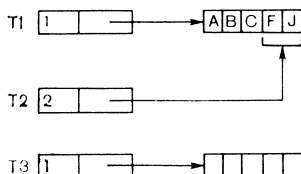
**З а м е ч а н и е.** В правой части присваивания текстовых ссылок не может использоваться текстовая константа, так как она не является текстовым объектом.

Присваивание текстовых значений заключается в переписывании литер текста из объекта, на который ссылается выражение, стоящее в правой части оператора присваивания в объект, на который ссылается переменная из левой части. В простейшем случае присваивание текстовых значений имеет следующий вид:

**ПЕРЕМЕННАЯ: = ТЕКСТОВОЕ ЗНАЧЕНИЕ**

Присваивание текстовых значений зависит от соотношения длин текстов левой и правой частей присваивания. Пусть длина текста в левой части присваивания равна ДЛ, а правой — ДП. Если  $ДЛ = ДП$ , то последовательно (слева направо) переписываются литеры из текста правой части в текст левой части. Если  $ДЛ > ДП$ , то переписываются все ДП литер в левую часть, а оставшиеся ДЛ — ДП литер заполняются пробелами. Если  $ДЛ < ДП$ , т. е. нельзя разместить все литеры из текста правой части в переменной из левой части, то выдается сообщение об ошибке. Указатели позиции в текстовых переменных остаются неизменными.

**Пример 3.5.** Пусть имеется структура данных:

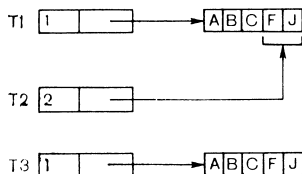


Тогда после выполнения следующих операторов:

$T2 := 'FJ',$

$T3 := T1,$

будет получена структура данных следующего вида:



Заметим, что в левой части оператора присваивания текстовых значений могут употребляться не только переменные, но и текстовые выражения. Например, занесение текста 'FJ' на место литер D и E в тексте T1 (пример 3.5) можно выполнить следующим оператором, в котором не используется переменная T2:

$T1.sub(4, 2) := 'FJ',$

**3.2.3. Сравнение текстовых ссылок и текстовых значений.** Текстовые значения можно сравнивать между собой с помощью операций сравнения значений:  $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $\geq$ ,  $>$ . Сравнение текстовых значений основано на использовании отношения литер, составляющих тексты. Два текста равны, если они либо оба пусты (равны *notext*), либо являются экземплярами одной и той же последовательности литер. Если эти условия не выполнены, то тексты считаются не равными. Текстовое значение T1 считается меньше, чем текстовое значение T2, только в том случае, если они не равны и выполнено одно из условий:

— T1 пусто;

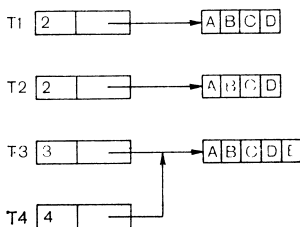
— вся последовательность литер из T1 совпадает с началом последовательности литер T2, но в T2 есть хотя бы еще одна литера;



— ранги первых  $K$  литер  $T1$  и  $T2$  равны, а ранг  $(K + 1)$ -й литеры из  $T1$  меньше, чем ранг  $K + 1$  литеры из  $T2$  (не совпадают могут ранги первых литер).

Кроме сравнения текстовых значений можно осуществлять проверку на идентичность (неидентичность) текстовых ссылок. Для сравнения текстовых ссылок используются знаки операций отношения ссылок  $==$ ,  $\neq$  (идентичность, неидентичность). Две ссылки на текст считаются идентичными, если они ссылаются на один и тот же текстовый объект, или обе равны **notext**. Отношение  $\neq$  является отрицанием отношения  $==$ .

**Пример 3.6.** Пусть имеется следующая логическая структура:



Для данной структуры отношения, приведенные ниже, имеют значение **true**:

$T1 = T2$   
 $T1 \neq T2$   
 $T3 > T2$   
 $T3 == T4$   
 $T1 \neq T3$

**З а м е ч а н и е.** При сравнении текстовых значений и текстовых ссылок указатели позиций во внимание не принимаются.

**3.2.4. Процедуры обработки текстов.** Вся работа с текстами осуществляется при помощи ряда системных (встроенных) процедур, которые можно рассматривать как атрибуты текстового объекта. Обращение к этим процедурам выполняется с помощью дистанционного доступа, аналогично тому, как при помощи дистанционных идентификаторов обращаются к атрибутам объекта. Слева от точки при обращении к атрибутам текста, указывается ссылка на текстовый объект. Если текстовая ссылка имеет значение **notext**, и используемая процедура не определена при таком значении, то выдается сообщение об ошибке. Ниже мы рассмотрим действия этих процедур.

**integer procedure length; ...;**

Значением этой процедуры является число литер в текстовом значении, на которое указывает текстовая ссылка. Если текстовая ссылка равна **notext**, то значением процедуры будет 0.

Приведем три процедуры, которые работают с указателем позиции.

**integer procedure pos; ...;**

Результатом процедуры является текущее значение указателя позиции. Например, непосредственно после исполнения оператора

**T:—text ('ABCDE');**

истинны следующие отношения

**T.length=5,**

**T.pos=1.**

Значение указателя позиции можно изменить с помощью процедуры

**procedure setpos (N); integer N; ...;**

При исполнении оператора **T.setpos (N)** указатель позиции текстовой ссылки **T** принимает значение фактического параметра. Если значение **N** выходит за диапазон (меньше 1 или больше **T.length**), то указателю присваивается значение **T.length+1**. Для проверки того, что значение указателя находится в указанном диапазоне, используется

**boolean procedure more; ...;**

Значение **T.more** истинно, если

**T.pos $\geq$ 1 и T.pos $\leq$ T.length**

Например, после исполнения операторов

**T:—text ('ABCDE');**

**T.setpos (7);**

**T.more** примет значение **false**. Данная процедура часто используется для проверки значения указателя позиции перед использованием процедур **getchar** и **putchar**.

Литеры текста доступны по одной. Для считывания литер из текста и записи литер в текст используются процедуры

**character procedure getchar; ...;**

**procedure putchar (C); character C; ...;**

Выбор литеры из текста производит процедура `getchar`. Она доставляет в качестве своего значения доступную (ту, на которую показывает указатель позиции) литеру. Запись литеры в текст производит процедура `putchar`. Она заменяет доступную литеру на значение фактического параметра. Каждая из процедур увеличивает на 1 значение указателя позиции.

**Пример 3.7.** Пусть заданы следующие описания и операторы:

```
text T, S;  
T:—text ('ШИФЕР');  
S:—text ('КАТЕР');  
S. putchar (T. getchar);  
T. putchar (S. getchar);
```

При выполнении оператора `S. putchar (T. getchar)` вначале вычисляется значение параметра `T. getchar`. Процедура `getchar` в качестве значения доставит первую литеру из текста `T`, т. е. "Ш". Значение указателя позиции для `T` увеличится на 1. Далее, эта литера вставляется на первую позицию в тексте `S`. Значение указателя позиции для `S` также увеличится на 1. После выполнения всех операторов

```
T. pos = S. pos = 3  
T = 'ШАФЕР'  
S = 'ШАТЕР'.
```

**3.2.5. Взаимные преобразования текстовых и арифметических значений.** Для преобразования текстовых значений в арифметические и обратного преобразования в языке предусмотрен ряд процедур, которые рассматриваются как атрибуты текста. Процедуры получения арифметических значений из текстов называются в языке симула-67 процедурами **дередактирования**, а процедуры преобразования арифметических значений в тексты — процедурами **редактирования**.

Рассмотрим процедуры **дередактирования**. Любая процедура **дередактирования** текста `T` выполняет следующие действия. Сначала определяется самая длинная последовательность литер, которая соответствует требуемому типу числа, содержащаяся в `T`. Найденная таким образом запись (последовательность) преобразуется в числовое значение. Указатель позиции принимает значение на единицу больше, чем последняя литера найденной записи. Если запись требуемого типа не будет найдена, то возникает ошибка.

Рассмотрим процедуры:

```
integer procedure getint; ...;
real procedure getreal; ...;
integer procedure getfrac; ...;
```

Процедура `getint` разыскивает в текстовом значении запись целого, т. е. последовательность цифр перед которой может стоять любое количество пробелов и знак. Значение, которое доставляется этой процедурой-функцией, равно целому значению, представленному данной записью.

**Пример 3.8.** Пусть задана последовательность операторов

```
text T, T1, T2;
T:—text ('__+1234.5+6.7E4КОНСТ');
T1:—T.sub (10, 8);
T2:—T.sub (14, 3);
```

Тогда

```
T. getint=1234      T. pos=8
T1. getint=6        T1. pos=3
```

Обращение `T2. getint` приведет к ошибке, так как в записи целого перед цифрами могут стоять только пробелы и знак числа. В данном случае перед цифрой стоит буква `E`.

Процедура `getreal` разыскивает в тексте запись целочисленного или вещественного значения с десятичной точкой и/или с указанием порядка, порядок представляется с помощью литеры `Q` или `E` при реализации на ЕС ЭВМ и `D` при реализации на БЭСМ-6.

Значение, которое доставляется этой процедурой-функцией, равно вещественному значению, представляемому найденной записью.

**Пример 3.9.** Пусть `T`, `T1`, `T2` определены как в примере 3.8. Тогда

```
T. getreal=1234.5      T. pos=10
T1. getreal=67000.0    T1. pos=7
T2. getreal=10000.0    T2. pos=3
```

Процедура `getfrac` разыскивает запись числа группами. Каждая группа за исключением, быть может, первой и последней состоит из трех цифр. Между группами находится разделитель — пробел. При интерпретации записи группами процедура игнорирует разделитель или десятичную точку, если таковые имеются. В качестве результата процедура доставляет значение целого.

**Пример 3.10.** Пусть `T`, `T1` определены как в примере 3.8, а `T2:—T.sub (13, 3)`. Тогда

T.getfrac=123      T.pos=7  
 T1.getfrac=67      T1.pos=5  
 T2.getfrac=7      T2.pos=2

Приведенные выше процедуры позволяют выделять из текста записи и интерпретировать их как арифметические значения.

Процедуры редактирования служат обратной цели — преобразованию арифметических значений в текстовые. Процедуры редактирования заполняют указанный текст T полученной записью числа. Отредактированная запись числа выравнивается по правому краю. Если в тексте T слева остается свободное место, то оно заполняется пробелами. Указатель позиции после редактирования принимает значение T.length+1. Если длина текста недостаточна, чтобы вместить результат редактирования, то происходит переполнение при редактировании и выдается сообщение об ошибке.

Имеются четыре процедуры редактирования:

**procedure putint (K); integer K; ...;**  
**procedure putfix (R, N); real R; integer N; ...;**  
**procedure putreal (R, N); real R; integer N; ...;**  
**procedure putfrac (K, N); integer K, N; ...;**

Процедура putint (K) преобразует значение параметра K в запись целого.

**Пример 3.11.**

**text T;**  
**T:—blanks (8);**

ОПЕРАТОР	ЗНАЧЕНИЕ T
T.putint (114)	'_____114'
T.putint (-19)	'_____—19'

Процедура putfix (R, N) редактирует значение R с учетом параметра N. Если  $N < 0$ , то во время исполнения оператора возникает ошибка. Если  $N = 0$ , то R преобразуется в запись целого, т. е.

**T.putfix (R, 0) = T.putint (R).**

Если  $N > 0$ , то R преобразуется в запись следующего вида:

**<ЗНАК> DDD.DDD ... DD**

Знак устанавливается только для отрицательных значений. Количество цифр после точки равно N.

**Пример 3.12.** Пусть переменная T определена, как в примере 3.11. Тогда

T.putfix (-8.53,5)	'— 8.53000'
T.putfix (____105.257,2)	'____105.26'

Использование T. putfix (—8.53,6) приведет к ошибке, так как для размещения такой записи необходимо место для 9 литер — '—8.530000'.

Процедура putreal (R, M) формирует запись числа следующего вида:

$\langle \text{ЗНАК} \rangle D.DD \dots DDE \langle \text{ЗНАК} \rangle DD$

Если  $M < 0$ , то во время исполнения оператора возникает ошибка. Если  $M = 0$ , то перед E устанавливается знак числа только в случае, если число отрицательное. Когда  $M = 1$ , то перед символом порядка ставится запись целого, состоящая из одной цифры. Если  $M > 1$ , то в дробной части записывается  $M - 1$  цифра.

Пример 3.13. T определена, как и в примере 3.11.

ОПЕРАТОР	ЗНАЧЕНИЕ T
T. putreal(—31.41, 0)	'___—E+01'
T. putreal(197.6, 1)	'___ 2E+02'
T. putreal(—0.215, 2)	' —2.2E — 01'

Процедура putfrac (I, N) формирует запись числа группами. Если  $N < 0$ , то возникает ошибка во время исполнения оператора. Если  $N = 0$ , то запись группами производится без точки, в противном случае устанавливается точка, за которой следует N цифр.

Пример 3.14. Пусть T:—blanks (12);

ОПЕРАТОР	ЗНАЧЕНИЕ T
T. putfrac (12345678, 0)	'__12__345__678'
T. putfrac (123456789, 5)	'1__234.567__89'
T. putfrac (—12345678, 3)	'__—12__345.678'

3.2.6. Передача текстов в качестве параметров. Текстовые ссылки и текстовые значения могут использоваться в качестве фактических параметров. В языке предусмотрены три способа передачи параметров: по значению, по ссылке, по наименованию. В случае, когда отсутствует указание о способе передачи, считается, что текст передается по ссылке. Для указания передачи по значению соответствующие формальные параметры должны быть указаны в списке значений, который присутствует в заголовке процедуры и начинается словом value. Если параметр передается по наименованию, то он указывается в списке имен, начинающемся словом name.

Передача по значению. Вычисление фактического параметра и присваивание полученного значения происходят в момент входа в процедуру или в момент порождения объекта. Этот способ передачи параметра эквивалентен выполнению оператора

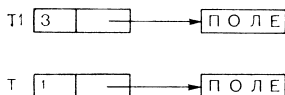
ФОРП :— text (ФАКП);

где ФОРП — формальный параметр, а ФАКП — фактический параметр. В результате выполнения этого оператора создается новый текстовый объект, локальный в теле процедуры (или класса). Длина текстового значения в этом объекте равна длине текста, являющегося фактическим параметром, а текстовое значение является копией текстового значения фактического параметра.

**Пример 3.15.** Рассмотрим следующий фрагмент программы.

```
text T1;
procedure F (T); value T; text T; ...;
T1:— text ('ПОЛЕ'); T1.setpos (3);
F (T1);
```

Ниже приведена логическая структура, возникающая при вызове процедуры F.



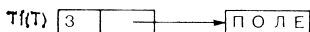
Заметим, что указатель позиции для T в процедуре F равен 1.

**Передача параметров по наименованию.** Вычисление фактического параметра происходит в момент использования этого параметра в теле процедуры. Передачу параметра по наименованию можно представить как текстуальную замену формального параметра фактическим.

**Пример 3.16.** Пусть заданы следующие описания и операторы:

```
text T1;
procedure F (T); name T; text T; ...;
T1:—text ('ПОЛЕ'); T1.setpos (3);
F (T1);
```

Ниже приведена логическая структура, получившаяся при вызове процедуры F перед началом исполнения ее операторов.



**Передача параметров по ссылке.** Вычисление фактического параметра происходит в момент обращения к процедуре или в момент порождения объекта. Передача параметра по ссылке эквивалентна выполнению оператора

## ФОРП:—ФАКП;

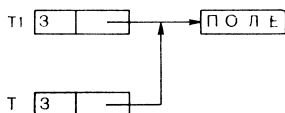
Очевидно, что текстовая константа не может в этом случае использоваться в качестве фактического параметра.

**З а м е ч а н и е.** В отличие от передачи параметров по наименованию, при передаче по ссылке значение указателя позиции для фактического параметра при выходе из процедуры равно значению указателя позиции перед входом в процедуру.

**П р и м е р 3.17.** Пусть заданы следующие описания и операторы:

```
text T1;  
procedure F (T); text T; ...;  
T1:—text ('ПОЛЕ'); T1.setpos (3);  
F (T1);
```

Ниже приведена логическая структура, получившаяся при вызове процедуры F перед началом исполнения ее операторов.



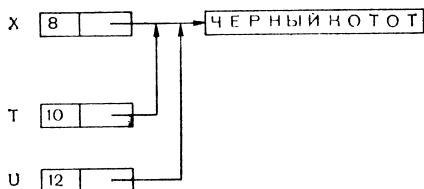
В заключение рассмотрим пример процедуры, которая реорганизует текст фактического параметра, переставляя литеры пробела в конец текста.

**П р и м е р 3.18.**

```
begin procedure СЖАТЬ (T); text T;  
begin text U; character C;  
T.setpos(1); U:—T;  
while U.more do  
begin C:=U.getchar;  
if C≠" " then T.putchar (C);  
end;  
L: while T.more do  
T.putchar (" ");  
end СЖАТЬ;  
text X; X:—text ('ЧЕРНЫЙ_КОТ');  
X.setpos (8); СЖАТЬ (X);  
end
```

Ниже представлена логическая структура, получившаяся перед выполнением оператора с меткой L.





$C = "T"$

После выполнения процедуры указатель позиции в X будет иметь то же значение, что и перед обращением к процедуре (8), а текстовое значение, на которое ссылается X, будет иметь следующий вид:

'ЧЕРНЫЙ КОТ\_\_'.

**З а м е ч а н и е.** В том случае, когда параметр T передается по наименованию, указатель позиции для X после выхода из процедуры будет равен 13.

**3.2.7. Программа обновления турнирной таблицы.** В этом разделе мы рассмотрим пример программы, осуществляющей внесение изменений в турнирную таблицу чемпионата страны по футболу. Общую структуру таблицы можно представить как набор строк, каждая из которых содержит порядковый номер команды, название команды, результаты игр данной команды на своем поле, количество побед, ничьих, поражений, количество забитых и пропущенных голов, очки и место. При построении таблицы используется тот факт, что чемпионат проходит в два круга: одна игра проходит на своем поле (дома) и одна на поле соперника (в гостях). Результат игры на чужом поле можно узнать из столбца, соответствующего порядковому номеру команды. Предполагается, что таблица находится на внешнем носителе. После очередного тура (игры) необходимо внести результаты в таблицу, выдать таблицу на печатающее устройство и запомнить обновленную таблицу на внешнем носителе. При написании программы используются, в основном, средства ввода-вывода языка симула-67 и ряд процедур работы с текстами.

В качестве исходной информации используется таблица с результатами предыдущих матчей, хранящаяся на внешнем устройстве в файле с именем FOOT, и результаты очередного тура, которые набиты на перфокартах. Результат игры набивается на перфокартах следующим образом: на первой перфокарте с первой позиции набивается название команды, играющей на своем поле, на второй перфокарте название команды-

соперника (тоже с первой позиции). На третьей перфокарте набивается результат встречи: голы, разделенные двоеточием. Например, «Спартак» играл матч в гостях с «Зенитом» и выиграл со счетом 2:0. Тогда на первой перфокарте набивается слово «Зенит», на второй — «Спартак», на третьей результат встречи — 0:2.

Будем представлять таблицу в виде текстового массива, где каждый элемент массива изображает строку таблицы. В этом случае удобно пользоваться встроенными в язык функциями, осуществляющими преобразование числовых значений в текстовые и обратные преобразования.

Программа должна уметь выполнять следующие функции:

- считывать таблицу с внешнего носителя;
- считывать информацию о состоявшемся туре и размещать ее в соответствующих строках таблицы;
- записывать измененную таблицу на внешний носитель и печатать новую таблицу на АЦПУ.

В дальнейшем остановимся на каждой функции подробнее. Описания необходимых переменных, если они еще не заданы, будут задаваться при рассмотрении каждой функции.

Как указывалось выше, таблицу будем представлять в виде текстового массива. Для считывания информации с внешнего файла необходимо определить файл (определение см. в 3.3.4), а также буфер ввода, через который осуществляется ввод. Для выполнения этих действий необходимо в программе иметь следующие описания и операторы:

```
integer K; ref (infile) ВВОД; text array T [1:18];  
comment число футбольных команд=18, определяем файл  
ввода;
```

```
(1) ВВОД:—new infile ('FOOT');
```

```
(2) ВВОД. open (blanks (127));
```

```
for K:=1 step 1 until 18 do
```

```
begin comment заполнение таблицы внешним файлом;
```

```
(3) T[K]:—blanks (127);
```

```
(4) T[K]:=ВВОД. intext (127);
```

```
end;
```

```
(5) ВВОД. close;
```

Оператор (1) определяет файл под именем FOOT, с которого будет осуществляться чтение турнирной таблицы. Следует отметить, что в управляющих картах задания необходимо указать карту (на ЕС ЭВМ — DD-предложение), идентифицирующую конкретное расположение этого файла во внешней памяти ЭВМ. Во втором операторе открывается файл и определяется размер записей, которые будут считываться из файла. В опе-

раторе (3) осуществляется порождение текстового объекта, состоящего из 127 символов, который соответствует содержимому строки таблицы. В следующем операторе в текстовое значение созданного объекта заносится содержимое буфера ввода. Так как размер буфера совпадает с размером текстового значения T[K], то при следующем запросе информации из файла, произойдет автоматическое заполнение буфера новой записью из внешнего файла. Оператор (5) закрывает файл чтения. Попытка обращения к этому файлу за новыми записями приведет к ошибке.

**З а м е ч а н и е.** Размер буфера ввода должен совпадать с размером считываемой записи.

Рассмотрим реализацию чтения информации о состоявшемся туре и размещения ее в строках таблицы. Для этого нам понадобятся некоторые дополнительные переменные:

**text** T1, T2; **integer** PE31, PE32, K1, K2;

В текстовых переменных T1 и T2 будут храниться названия команд, между которыми состоялась встреча. В переменных PE31 и PE32 указывается результат встречи. Переменные K1 и K2 соответствуют строкам таблицы, в которых указана информация о командах, сыгравших между собой. Каждая строка информации разбита на позиции. Позиции строки 8—18 отводятся под название команды. Начиная с 18-й позиции, отводятся группы по 3 позиции под результат матча, который записывается в виде тройки N : K, где N — число забитых, K — число пропущенных голов. В рассматриваемых операторах указываются конкретные значения позиций строк таблицы. В 93—94, 98—99, 103—104 позициях соответственно указывается число побед, ничьих и поражений, в 108—109, 111—112 — забитые и пропущенные голы, в 116—117 — количество очков.

Для изменения значений, находящихся в строке таблицы, используется процедура ЗАП:

**procedure** ЗАП (N, K, J); **integer** N, K, J;  
T[N].sub (K, 2).putint (T[N].sub (K, 2).getint+J);

Процедура ЗАП выбирает из строки N таблицы подтекст, состоящий из 2-х позиций, начиная с K-й позиции. Полученный текст преобразуется в целое с помощью процедуры getint. Далее к данному целому прибавляется значение J. Полученный результат преобразуется в запись целого, используя процедуру putint, и помещается на прежнее место (т. е. в строку N, начиная с K позиции). Ниже приведен фрагмент программы, реализующий чтение информации с перфокарт и располагающий ее в таблице.

```

(1)      T1:—blanks (10); T2:—blanks (10);
(2) ТУР: if lastitem then goto КОНЕЦ ТУРА;
(3)      T1:=intext (10); inimage;
(4)      T2:=intext (10); PE31:=inint; PE32:=inint;
(5)      for K:=1 step 1 until 18 do
(6)      begin if T[K].sub (8, 10)=T1 then K1:=K;
(7)          if T[K].sub (8, 10)=T2 then K2:=K;
           end;
           comment заполнение строк таблицы;
(8)      if PE31>PE32 then
(9)      begin ЗАП (K1, 93, 1);
(10)         ЗАП (K2, 103, 1);
(11)         ЗАП (K1, 116, 2);
           end else
           if PE31=PE32 then
(12)      begin ЗАП (K1, 98, 1);
(13)         ЗАП (K2, 98, 1);
           if T[K1].sub (98, 2).getint lt 10 then
(14)         ЗАП (K1, 116, 1);
           if T[K2].sub (98, 2).getint lt 10 then
(15)         ЗАП (K2, 116, 1);
           end else
(16)      begin ЗАП (K1, 103, 1);
(17)         ЗАП (K2, 93, 1);
(18)         ЗАП (K2, 116, 2);
           end;
(19)      ЗАП (K1, 108, PE31);
(20)      ЗАП (K1, 111, PE32);
(21)      ЗАП (K2, 108, PE32);
(22)      ЗАП (K2, 111, PE31);
(23)      T[K1].setpos (20+(K2-1)*4);
(24)      T[K1].sub (T[K1].pos, 1).putint (PE31);
(25)      T[K1].setpos (T[K1].pos+1),
(26)      T[K1].sub (T[K1].pos, 1).putchar (" ");
(27)      T[K1].sub (T[K1]. pos, 1).putint (PE32);
(28)      goto ТУР;
           КОНЕЦ ТУРА:

```

Рассмотрим назначение операторов приведенного фрагмента программы. Операторы (1) создают текстовые объекты, в которых будут храниться названия команд, сыгравших между собой. В следующем операторе проводится проверка на окончание ввода с перфокарт. Если все литеры на оставшихся перфокартах являются пробелами, то выполнение данного фрагмента программы завершается (уход на метку КОНЕЦ ТУРА).

В противном случае переходим к выполнению операторов третьей строки. Первый оператор строки заполняет текстовое значение литерами с перфокарты. Следующий оператор (*inimage*) заполняет буфер образом следующей перфокарты. Если бы этот оператор отсутствовал, то в T2 записались бы литеры пробела. Первый оператор в четвертой строке заполняет текстовое значение T2 названием второй команды. Оставшиеся операторы вводят результат игры и размещают его в переменных PE31 и PE32. Заметим, что в данном случае здесь не указан оператор *inimage*. Это вызвано тем, что процедура *inint* разыскивает в буфере запись целого и если в буфере такой записи нет, то происходит заполнение буфера образом очередной карты. При дальнейшем вводе перед первым оператором *intext* не надо ставить *inimage*, так как вначале будет выполняться процедура *lastitem*, которая сама пропустит все пробелы и введет образ очередной карты в буфер. В операторах (6) — (7), в цикле, осуществляется сравнение названия рассматриваемых команд с названиями команд из таблицы и присваивание переменным K1 и K2 значений номеров соответствующих строк таблицы. Для этого используется функция *sub*, которая выбирает из текстового объекта T[K] название команды и сравнивает его с значением, введенным с перфокарты.

Далее возникают три возможности:

1)  $PE31 > PE32$  (команда, играющая дома, одержала победу),

2)  $PE31 = PE32$  (игра закончилась ничейным результатом),

3)  $PE31 < PE32$  (команда, играющая дома, проиграла).

Случаи 1) и 3) симметричны, так что рассмотрим один из них, а именно 1). Команда, играющая дома, победила. В этом случае необходимо:

а) увеличить число побед в графе «победы» для команды, играющей дома (номер строки — K1),

б) увеличить число очков в графе «очки» для команды, играющей дома,

в) увеличить число поражений для команды, играющей в гостях (номер строки K2).

Все эти действия выполняются с помощью трех операторов вызова процедур (9) — (11). Для случая 3) эти же действия выполняют операторы (16) — (18). Во втором случае (ничья) в графе «ничья» в строках K1 и K2 значение увеличивается на 1. Перед увеличением количества очков необходимо проверить, не превосходит ли количество ничьих 10 (по положению такие игры не приносят очков). Если нет, то количество очков увеличивается на 1. В операторах (19) — (22) увеличивается значение забитых и пропущенных голов для команд.

Последнее, что необходимо сделать, это внести результат встречи в таблицу. Как говорилось ранее, результат встречи заносится в строку, соответствующую команде, играющей на своем поле, и в столбец, номер которого соответствует номеру команды противника. В операторе (23) в строке K1 таблицы указатель позиции устанавливается с помощью процедуры `setpos` на соответствующее команде K2 место. Оператор (24) заносит в это место значение числа забитых голов. Значение указателя позиций для K1 строки не изменилось, так как применяется функция `putint` для подтекста текста T[K1]. Поэтому необходимо увеличить указатель позиций для T[K1] строки на 1 (оператор (25)). Оператор (26) помещает литеру ":" в текущую позицию, как это обычно принято при заполнении таблицы. Процедура `putchar` сама продвигает в конце своей работы указатель позиции на 1. Оператор (27) помещает на это место число пропущенных голов. Оператор (28) осуществляет переход на начало данного фрагмента для считывания информации о других встречах тура.

После работы частей, рассмотренных выше, необходимо распечатать таблицу на АЦПУ и записать ее на прежнее место (в файл FOOT). Для этого требуется дополнительная переменная, указывающая на файл вывода.

**ref (outfile) ВЫВОД;**

(1) **ВЫВОД:**—new outfile ('FOOT');

(2) **ВЫВОД.open** (blanks (127));

**for** K:=1 **step** 1 **until** 18 **do**

(3) **begin** **ВЫВОД.outtext** (T[K]);

(4) **outtext** (T[K]);

**end;**

(5) **ВЫВОД.close;**

Эта часть программы во многом схожа со чтением информации с внешнего устройства. Операторы (1)—(2) определяют внешний файл и буфер вывода. Оператор (3) осуществляет последовательную запись строк таблицы на внешний файл. Оператор (4) производит печать строк таблицы на АЦПУ. Заметим, что строки будут размещаться на АЦПУ одна под одной, так как размер выводимой строки равен размеру строки АЦПУ. В противном случае нам пришлось бы использовать оператор `outimage`, который выталкивает буфер вывода на АЦПУ. Оператор (5) закрывает возможность дальнейшего вывода на этот файл. Если в момент закрытия буфер файла оказался не пуст, то производится выталкивание последнего образа на внешний файл.

В данном примере, конечно же, не раскрываются все возможности ввода-вывода и работы с текстами. Кроме того, мы не рассматривали здесь вопросы сортировки строк в зависимости от количества очков и т. д. Выдачу таблицы можно снабдить заголовком. В данном случае, это, на наш взгляд, не является существенно важным и не вызовет у желающих это сделать каких-либо дополнительных трудностей.

### 3.3. Ввод-вывод в языке симула-67

Средства ввода-вывода позволяют выполнять взаимные преобразования данных типа `real`, `integer`, `text`, `character` в последовательности символов, которые могут участвовать в обмене с внешними устройствами. Эти преобразования выполняются с помощью соответствующих встроенных процедур редактирования и дередктирования, определенных в языке симула-67. Форматы преобразований описываются значениями фактических параметров этих процедур.

**3.3.1. Структурная организация ввода-вывода.** Процессы ввода-вывода в языке основаны на понятии *файл*. Под файлом будем понимать последовательную структуру данных, внешних по отношению к программе. Каждый файл состоит из записей. Запись — упорядоченная последовательность литер определенной длины. Примером файла может служить колода перфокарт, в которой карта является записью.

Все средства, с помощью которых проводится обмен информацией с внешними устройствами, определены как атрибуты системного класса `basicio`. Программа пользователя считается погруженной в блок с префиксом `basicio`, т. е. в ней непосредственно доступны все его атрибуты.

В классе `basicio` определены четыре типа файлов:  
`infile` — последовательный файл ввода информации,  
`outfile` — последовательный файл вывода информации,  
`printfile` — последовательный файл вывода на АЦПУ,  
`directfile` — файл ввода-вывода с прямым доступом.

`Directfile` в дальнейшем рассматриваться не будет. Концепции ввода-вывода, определяющие файлы заданного типа, определены в системных классах, описания которых заданы в классе `basicio`. Таким образом, связь программы с внешним файлом осуществляется через объект определенного класса, важным атрибутом которого является текстовая переменная, используемая как буфер ввода-вывода.

Для того чтобы начать работу с внешним файлом, необходимо сначала сгенерировать объект соответствующего класса, определяющий свойства файла: для чтения информации созда-

ется объект класса infile, а для записи — объект класса outfile. Заметим, что в системном классе basicio сгенерированы объект класса infile для ввода с перфокарт и объект класса printfile для вывода информации на АЦПУ, и поэтому в программе не надо генерировать эти объекты.

Атрибутом каждого файла является имя данного файла. Способы задания имени зависят от конкретных реализаций языка. При выполнении программ, использующих внешние файлы, необходимо в паспорте задания предусмотреть карты, связывающие имя файла с физическим файлом на внешнем устройстве (набором данных).

В объектах, осуществляющих связь с внешними файлами каждого типа, имеется текстовый атрибут image. В последовательных файлах переменная image служит для ссылки на текстовый объект, который выступает в роли буфера, т. е. содержит обрабатываемую в данный момент запись. Длина текста image соответствует размеру записи и определяется при открытии файла. При выводе информации на внешний файл производится постепенное заполнение буфера вывода, причем арифметические значения преобразуются в тексты с помощью процедур редактирования. При заполнении буфера его содержимое автоматически переносится на внешний файл, а сам буфер освобождается для приема следующей порции информации, т. е. его указатель позиции устанавливается в 1, а сам он заполняется пробелами. С помощью системной процедуры outimage можно принудительно вывести содержимое буфера и освободить его. Ввод информации проводится аналогичным образом: считанная из внешней памяти запись помещается в буфер image в текстовом виде. Соответствующие числовые значения получаются с помощью процедур доредактирования.

Перед выполнением операций обмена с файлом его необходимо открыть. Открытие файла производит процедура open, являющаяся атрибутом соответствующего файла. После завершения работы с файлом требуется его закрыть с помощью процедуры close.

Ниже приведена общая схема класса basicio.

```
class basicio (N); integer N;
begin ref (infile) SYSINP; ref (printfile) SYSOUTP;
      ref (infile) procedure sysin; sysin:—SYSINP;
      ref (printfile) procedure sysout; sysout:—SYSOUTP;
      class file; . . . . . ;
      file class infile; . . . . . ;
      file class outfile; . . . . . ;
      outfile class printfile; . . . . . ;
```



```

SYSINP:—new infile ('SYSIN');
SYSOUTP:—new outfile ('SYSOUT');
SYSINP.open (blanks(80));
SYSOUTP.open (blanks (N));
inner;
sysin.close;
sysout.close;
end  BASICIO;

```

Объекты каждого класса из перечисленных в *basicio* имеют ряд общих атрибутов для организации ввода-вывода. Эти атрибуты описаны в классе *file*, который используется в качестве префикса к классам, описывающим различные типы файлов. К этим атрибутам относятся процедуры *open*, *close*, *setpos*, *pos*, *more*, *length* и текстовая переменная *image*. Эти процедуры включены в класс *file* для обеспечения удобства при работе с файлами. Эти процедуры применяются всегда только к текстовому объекту, на который ссылается переменная *image*. Описания классов для каждого типа файлов содержат, кроме средств, описанных в классе *file*, дополнительные процедуры по редактированию (дередактированию) информации в буфере *image*.

Программа пользователя ведет себя так, как если бы она имела следующую структуру:

```

basicio (N) begin inspect SYSINP do
    inspect SYSOUTP do
        <программа>
    end
end

```

Рассмотрим, какие действия по организации ввода-вывода выполняются перед началом исполнения программы. Параметр *N* указывает размер буфера для файла, отображающего вывод информации на АЦПУ. В реализациях на БЭСМ-6 и ЕС ЭВМ значение *N* равно 127. Кроме того, в классе *basicio* выполняются действия по генерации объектов класса *infile* (ввод с перфокарт) и класса *printfile* (вывод на АЦПУ) и открытию этих файлов. Ссылки на эти объекты доступны пользователю через процедуры *sysin* и *sysout*. При завершении программы пользователя управление возвращается в класс *basicio*, в котором проводится закрытие файлов.

**3.3.2. Определение файлов в симула-программе.** Связь с внешними файлами в симула-программе осуществляется с помощью создания объектов, отражающих свойства файлов. Все процедуры обмена с файлами определены как атрибуты соответствующих им объектов и доступны благодаря дистанционным обращениям. Слева от точки в таких обращениях должна стоять ссылка на объект, определяющий

файл. Для хранения ссылки на объект в программе необходимо описать ссылочную переменную, квалифицированную либо классом `infile` для файлов чтения, либо классом `outfile` для файлов записи. Ссылки на системные файлы ввода-вывода можно получить с помощью процедур-функций `sysin` (ввод с перфокарт) и `sysout` (вывод на устройство печати), определенных в классе `basicio`.

Параметром всех классов, описывающих работу с внешними файлами, является имя файла. По данному параметру устанавливается связь между конкретным набором данных на внешнем устройстве и соответствующим ему объектом в симула-программе. Способы создания имени могут зависеть от вычислительной системы, на которой реализуется язык.

Внешний файл на БЭСМ-6 идентифицируется с помощью математического номера устройства и начального номера зоны, с которой располагается данный файл. Этим устройством может быть лента, диск и т. п.

**Пример 3.19.** Пусть необходимо определить файл вывода с именем 'ВЫВОД НА МЛ' на ленте 1118, начиная с 120 (в десятичной системе счисления) зоны. В этом случае следует сделать следующее. В паспорте задачи необходимо поставить карту заказа магнитной ленты 1118 с математическим номером, предположим, 42:

ЛЕНТ 42(1118—ЗП)

для хранения ссылки на объект требуется описать ссылочную переменную, например:

`ref (outfile) ВЫВОД;`

Оператор генерации объекта файла вывода тогда будет иметь следующий вид:

`ВЫВОД:— new outfile ('ВЫВОД НА МЛ', 42, 120);`

На ЕС ЭВМ именем файла, задаваемым при генерации соответствующего объекта, служит имя DD-предложения, указывающего расположение массива записей во внешней памяти. Это DD-предложение должно быть задано на шаге задания, выполняющего счет симула-программы. В качестве имени файла в симула-программе можно использовать текстовую переменную, которая в момент открытия файла должна иметь значение, отличное от `notext`, причем его первые 8 символов должны быть допустимыми для DD-предложений языка управления заданиями в ОС ЕС.

**Пример 3.20.** Пусть необходимо определить файл 'INPUT' для ввода из последовательного набора данных, расположенного на диске с именем MD0001.

Для этого необходимо ввести ссылочную переменную  
**ref (infile) ВВОД;**

**и оператор**

**ВВОД:—new infile ('INPUT');**

Для идентификации данного файла можно использовать следующее DD-предложение:

```
//INPUT DD DSN=EXPDATA, DISP=SHR,  
// VOL=SER=MD0001, UNIT=5050
```

В главе 4 (п. 2.2) дается подробное описание особенностей средств ввода-вывода, реализованных в симула-трансляторах для БЭСМ и ЕС ЭВМ, приводятся примеры определения файлов и работы с ними.

После того как объект заданного типа файла сгенерирован, можно использовать все атрибуты данного файла. Для этого достаточно иметь ссылку на данный объект. Обращение к атрибуту объекта будет иметь вид

*⟨ссылка на объект, соответствующий файлу⟩.⟨атрибут⟩*

Как отмечалось выше, программа пользователя погружена в подразумеваемые присоединяющие операторы, дающие непосредственный доступ к системным файлам **sysin** и **sysout**. Поэтому в обращениях к атрибутам файла ввода с перфокарт и файла вывода на АЦПУ можно опускать ссылку на эти объекты. Следует отметить, что в данном случае существует конфликт имен **open**, **close**, **image**, **setpos**, **pos**, **more**, **length** для файлов ввода с перфокарт и вывода на АЦПУ. Разрешение данного конфликта происходит с учетом операторов присоединения. Когда такой конфликт возникает, то данные имена связываются с объектом **sysout**. Соответствующие атрибуты для **sysin** получаются с помощью обращений: **sysin.image**, **sysin.setpos** и т. п.

**3.3.3. Открытие файла.** Ранее мы описали средства для определения файлов. Прежде чем приступить к обмену информацией с внешней памятью необходимо открыть файл. Открытие файла выполняется процедурой **open (T)**. Параметром **T** данной процедуры является ссылка на текстовый объект, который будет выполнять роль буфера.

Схема процедуры **open** имеет следующий вид:

```
procedure open (T); text T;  
begin if ОТКРЫТ then ОШИБКА;  
ОТКРЫТ:=true;  
image:—T;  
. . . . .  
end open;
```

При вызове процедуры `open` сначала проверяется, открыт файл или нет. Если файл уже был открыт, то фиксируется ошибка. В противном случае устанавливается признак открытого файла. Далее переменной `image` присваивается ссылка на текстовый объект, на который ссылается фактический параметр `T`. Размер записи открываемого файла равен длине фактического параметра. Напомним, что пользователь не должен открывать файлы `sysin` и `sysout`, которые открываются в системном классе `basicio`.

Процедура `open`, определенная в классе `infile`, заполняет текстовое значение буфера образом первой записи файла. При реализации языка на ЕС ЭВМ и БЭСМ-6 текстовые значения буферов для системного файла ввода `sysin` и системного файла вывода `sysout` при открытии заполняются литерами пробела.

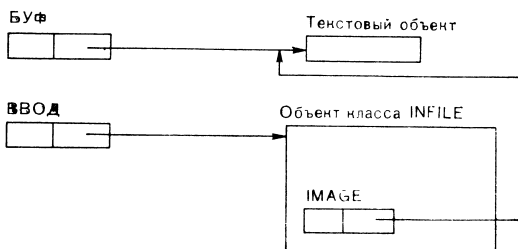
**Пример 3.21.** Пусть при работе на ЕС ЭВМ необходимо открыть файл, определенный в примере 3.20. Предполагается, что набор данных `EXPDATA` состоит из 80-байтовых записей.

**text БУФ;**

(1) БУФ:—blanks (80);

(2) ВВОД.open (БУФ);

В операторе (1) генерируется текстовый объект, который будет играть роль буфера. Оператор (2) открывает файл ввода. Ниже приведена логическая структура, возникающая в результате открытия файла.



В текстовом значении после выполнения операции открытия файла будет размещена первая запись из внешнего файла.

**Замечание.** В качестве буфера не допускается использование подтекста.

**3.3.4. Процедуры ввода.** Ввод данных в языке выполняется с помощью процедур ввода, определенных в системном классе `infile`. В этом классе имеется 6 процедур-функций ввода, а именно `inimage`, `inchar`, `inint`, `inreal`, `infrac`, `intext`, и две процедуры-функции `endfile` и `lastitem` для провер-

ки на окончание файла. Все процедуры работают над текстом, находящимся в буфере ввода, или над его частью (подтекстом).

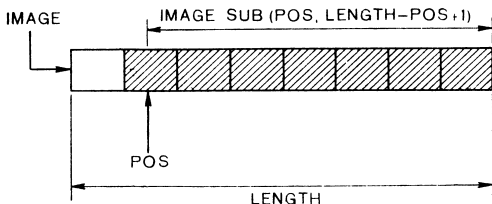


Рис. 24. Подтекст, обрабатываемый процедурами ввода.

Процедуры ввода берут информацию из части буфера ввода, ограниченной слева литерой, на которую указывает значение `pos`, а справа — концом текста (рис. 24). Каждая из процедур ввода продвигает указатель позиции, если при ее выполнении не будет обнаружена ошибка во вводимой информации. В приводимых ниже примерах будет определяться значение указателя позиции, которое он принимает после выполнения соответствующей процедуры ввода.

Когда указатель позиции превысит длину буфера ввода, производится ввод следующего образа с внешнего файла с помощью процедуры `inimage`:

```
procedure inimage; ...;
```

Процедура `inimage` помещает в буфер ввода очередной образ и ставит указатель позиции в 1. Если внешних образов больше нет, то буфер ввода заполняется пробелами и выдается сообщение об ошибке.

```
boolean procedure endfile; ...;
```

Процедура `endfile` доставляет значение `true`, если файл ввода закрыт или если файл открыт, но была прочитана последняя запись файла.

```
boolean procedure lastitem; ...;
```

Процедура `lastitem` доставляет значение `true`, если файл ввода закрыт или если файл открыт, но все оставшиеся до конца файла литеры являются литерами пробела.

```
character procedure inchar; ...;
```

Для ввода одной литеры текста используется процедура `inchar`. Эта процедура доставляет в качестве значения литеру текста, на которую указывает `pos`, и увеличивает значение `pos` на единицу.

**text procedure intext (N); integer N; ...;**

Эта процедура служит для ввода текста длиной N, причем значение N должно лежать в интервале  $1 \leq N \leq \text{image.length}$ . Если N выходит за допустимый интервал, то выдается сообщение об ошибке и `pos` не изменяется. Если значение N больше длины подтекста, оставшегося в буфере, то подтекст игнорируется, происходит обращение к процедуре `inimage`, после чего производится ввод текста с обновленного значения буфера.

Ввод целых и вещественных чисел осуществляется с помощью процедур ввода `inint`, `inreal`, `infrac`. Все эти процедуры набирают в подтексте буфера ввода запись числа требуемого вида. Если в подтексте буфера ввода все литеры являются пробелами, то подтекст игнорируется, происходит обращение к процедуре `inimage`. Набор записи числа в этом случае происходит с обновленного значения буфера. Набор заканчивается, когда очередная литера противоречит искомому виду. Все начальные литеры, отличные от литер, допустимых для записи числа требуемого вида, игнорируются. Если в результате редактирования записи числа полученное значение выходит за диапазон представления чисел, то выдается сообщение об ошибке. Значение, получающееся в результате выполнения процедур ввода, в этом случае неопределено.

**З а м е ч а н и е.** Максимальное представимое целое число для реализации на ЕС ЭВМ — 2147483647, на БЭСМ-6 — 1099511627775. Диапазон абсолютных величин (A) действительных чисел для реализации на ЕС ЭВМ составляет  $2.4 \cdot 10^{-78} \leq A \leq 7.2 \cdot 10^{75}$ , БЭСМ-6 —  $2^{-64} \leq A \leq 2^{63}$ .

После выполнения процедур ввода чисел указатель позиции становится на единицу больше, чем номер последней литеры найденной записи числа. Если запись числа требуемого вида не найдена, то выдается сообщение об ошибке, `pos` не изменяется, а в качестве значения процедуры выдается 0.

Рассмотрим на примерах работу процедур ввода чисел в предположении, что текстовые представления чисел находятся на устройстве ввода перфокарт.

**integer procedure inint; ...;**

Процедура предназначена для ввода целых чисел. Значение, доставляемое этой процедурой-функцией, равно целому числу, представляемому найденной записью целого.

**П р и м е р 3.22.** Пусть на вводимой карте, начиная с первой позиции, находится следующая информация:

NA=25 NB=-3.52 NB=2E+01 NG=12345678912345

Тогда при выполнении следующих операторов:

**for K:=1 step 1 until 6 do**

**N:=inint**

целой переменной N и указателю позиции будут присваиваться соответственно значения

ЗНАЧЕНИЕ K	ЗНАЧЕНИЕ N	ЗНАЧЕНИЕ УКАЗАТЕЛЯ ПОЗИЦИИ
1	25	6
2	—3	12
3	52	15
4	2	20
5	1	24
6	Не определено	42

**real procedure inreal; . . .;**

Данная процедура предназначена для ввода вещественных чисел.

Значение, доставляемое этой процедурой-функцией, равно вещественному значению, представляемому найденной записью вещественного числа. В качестве показателя степени вещественных чисел используется символ  $\propto$  или E при реализации на ЕС ЭВМ и  $\diamond$  при реализации на БЭСМ-6.

**Пример 3.23.** Пусть на вводимой карте, начиная с первой позиции, располагается следующая информация:

NA=—3.52 NB=E—4 NB=1.5E0017 NG=.532 $\propto$ +2  
ND=53 NE=2E+76

Тогда при выполнении следующих операторов:

**for K:=1 step 1 until 6 do**

**A:=inreal;**

вещественной переменной A и указателю позиции будут присваиваться значения

ЗНАЧЕНИЕ K	ЗНАЧЕНИЕ A	ЗНАЧЕНИЕ УКАЗАТЕЛЯ ПОЗИЦИИ
1	—3.5200000E+00	9
2	1.0000000E—04	16
3	1.5000000E+17	28
4	5.3200000E+01	39
5	5.3000000E+01	45
6	Не определено	54

**integer procedure infrac; . . .;**

Эта процедура предназначена для ввода целых чисел, записанных особым образом в виде так называемой записи группами. При интерпретации записи группами процедура игнорирует одиночные пробелы между группами цифр и десятичную точку, если таковые имеются. Значение, доставляемое этой процедурой-функцией, равно получающемуся таким образом целому значению. Диапазон допустимых значений такой же, как и у процедуры `inint`.

**Пример 3.24.** Пусть на вводимой карте, начиная с первой позиции, находится следующая информация:

NA=12345 NB=4.56 NB=3 56.258 NG=-3.249.41  
ND=5 678.25

Тогда при выполнении следующих операторов:

```
for K:=1 step 1 until 8 do
  N:=infrac;
```

целой переменной `N` и указателю позиции будут присваиваться значения

ЗНАЧЕНИЕ K	ЗНАЧЕНИЕ N	ЗНАЧЕНИЕ УКАЗАТЕЛЯ ПОЗИЦИИ
1	123	7
2	45	9
3	456	17
4	3	22
5	56258	29
6	-3249	39
7	41	42
8	567825	54

**3.3.5. Процедуры вывода.** Вывод данных в языке осуществляется с помощью процедур вывода, определенных в системном классе `outfile`. Имеется 7 процедур вывода: `outimage`, `outchar`, `outint`, `outfrac`, `outfix`, `outreal`, `outtext`.

Эти процедуры осуществляют передачу данных из программы в буфер вывода, выполняя, если это необходимо, преобразование данных в текстовую форму. В процессе передачи происходит постепенное заполнение информацией буфера вывода. Когда текст, представляющий очередное значение выводимой величины, не может быть размещен в оставшейся части буфера, процедуры вывода обращаются к процедуре `outimage`, которая осуществляет передачу очередного текстового образа, содержащегося в буфере, в выходной файл на внешнем устройстве. После выдачи очередного образа процедура `outimage` заполняет буфер пробелами и устанавливает значение указателя



позиции pos равным 1, а обратившаяся к ней процедура вывода размещает непоместившийся текст с первой позиции буфера.

В языке симула-67 в системном классе `basicio` определены процедуры для вывода целых, вещественных, литерных и текстовых значений. Каждое число с помощью процедур редактирования преобразуется в некоторый подтекст буфера вывода. На первую литеру этого подтекста указывает указатель позиции буфера, а длина подтекста равна значению параметра `L`, который явно указывается в процедурах вывода чисел. В качестве значения параметра `L` процедуры `outtext(T)` берется длина текста `T`. В роли `T` может быть строка или текстовое выражение.

Значение параметра `L` должно быть больше 0. Если значение параметра `L` меньше, чем длина записи числа, полученная после редактирования числа, то выдается сообщение об ошибке, в противном случае полученная запись числа выравнивается в подтексте вправо и дополняется пробелами до `L` позиций. В случае, когда значение `L` превышает длину подтекста, оставшегося до конца буфера, перед редактированием происходит обращение к процедуре `outimage`. Если в процедуре вывода значение `L` превышает длину буфера вывода, то выдается сообщение об ошибке, и результат выполнения процедуры эквивалентен действию процедуры `outimage`.

**procedure** `outtext (T); value T; text T; ...;`

Процедура `outtext` служит для вывода текстового значения, на которое указывает фактический параметр (если в качестве фактического параметра используется текстовая ссылка). Если в качестве параметра указано текстовое значение, то выводится значение фактического параметра.

Вывод целых и вещественных чисел выполняется с помощью процедур: `outint (K, L)`, `outfrac (K, N, L)`, `outfix (A, N, L)`, `outreal (A, N, L)`. Все эти процедуры осуществляют преобразования арифметических значений в записи чисел с последующей выдачей их в буфер вывода. После выполнения процедур указатель позиции буфера увеличивается на значение параметра `L`. Если при выполнении процедур были обнаружены ошибки, указатель позиции не изменяется. Исключение составляет ошибка, вызванная тем, что длина текста `L`, в который производится запись, недостаточна, чтобы вместить выводимый текст или результат редактирования числа.

**З а м е ч а н и е.** Если значение параметра `N` у процедур `outfrac`, `outfix`, `outreal` меньше нуля, то выдается сообщение об ошибке, редактирование числа не производится. В буфер вы-

вода помещается L пробелов, и указатель позиции буфера увеличивается на значение параметра L.

Положительное число преобразуется в запись числа без знака, отрицательное число — в запись числа со знаком минус, который непосредственно предшествует первой значащей цифре. Ведущие незначащие нули подавляются, кроме нуля, непосредственно предшествующего десятичной точке.

Рассмотрим на примерах работу процедур вывода чисел в предположении, что вывод осуществляется на системное устройство вывода (АЦПУ).

**procedure outint (K, L); integer K, L; ...;**

Процедура предназначена для вывода целого числа. Значение целого параметра K преобразуется в последовательность из L символов, представляющую это значение.

**Пример 3.25.** Пусть целой переменной K присвоено значение 12345678. Используя операторы outint (K, L) с различными значениями параметра L, получим следующие записи целых чисел:

ОПЕРАТОР	image. pos	ЗАПИСЬ ЦЕЛОГО
1 outint(K, 15)	16	12345678
2 outint(K, 8)	24	12345678
3 outint(K, 5)	29	1234?

В последнем случае будет выдано сообщение об ошибке, а последним знаком в поле будет вопросительный знак в реализации на ЕС ЭВМ и ↑ на БЭСМ-6.

**procedure outfrac (K, N, L); integer K, N, L; ...;**

Процедура предназначена для вывода целого числа в особой форме — группами. Каждая группа, за исключением, быть может, первой и последней, состоит из трех цифр. Между группами помещается разделитель — пробел.

Если значение параметра  $N > 0$ , то формируется запись группами с десятичной точкой, за которой следует N цифр. Если  $N = 0$ , то формируется запись группами без десятичной точки.

**Замечание.** При реализации этой процедуры введено ограничение на параметр N. При реализации на ЕС ЭВМ значение параметра N должно удовлетворять условию  $0 \leq N \leq 11$ , при реализации на БЭСМ-6 —  $0 \leq N \leq 14$ . Если N выходит за допустимый диапазон, то выдается предупреждающее сообщение, редактирование числа производится с параметром  $N = 11$  (для ЕС ЭВМ) или  $N = 14$  (для БЭСМ-6) и указатель позиции буфера увеличивается на значение L.

**Пример 3.26.** Пусть целая переменная  $K$  имеет значение 2567895. Используя операторы  $\text{outfrac}(K, N, L)$  с различными значениями параметров  $N$  и  $L$ , получим следующие записи группами:

ОПЕРАТОР	image. pos	ЗАПИСЬ ГРУППАМИ
1 $\text{outfrac}(K, 0, 10)$	11	2 567 895
2 $\text{outfrac}(K, 2, 11)$	22	25 678.95
3 $\text{outfrac}(K, 3, 10)$	32	2 567. 895
4 $\text{outfrac}(K, 5, 9)$	41	25.678 95
5 $\text{outfrac}(K, 15, 20)$	61	.000 025 678 95
6 $\text{outfrac}(K, -1, 6)$	67	
7 $\text{outfrac}(K, 5, 3)$	70	25?

При выполнении оператора под номером 5 выдается предупреждающее сообщение. При выполнении оператора под номером 6 будет выдано сообщение об ошибке и редактирование числа не произойдет.

**procedure** outfix ( $A, N, L$ ); **real**  $A$ ; **integer**  $N, L$ ; ...;

Процедура предназначена для вывода вещественных чисел. В зависимости от значения параметра  $N$  формируется различная запись числа. Если значение параметра  $N=0$ , то действие процедуры аналогично выполнению процедуры  $\text{outint}$ . В случае, когда  $N>0$ , формируется запись мантиссы, у которой в дробной части содержится  $N$  цифр. Получающаяся запись мантиссы обозначает число, равное значению параметра  $A$ , округленному до  $N$ -го десятичного знака.

**Пример 3.27.** Пусть вещественной переменной  $A$  присвоено значение  $-325.62791$ . Используя операторы  $\text{outfix}(A, N, L)$  с различными значениями  $N$  и  $L$ , получим следующие записи мантиссы:

ОПЕРАТОР	image. pos	Запись Мантиссы
1 $\text{outfix}(A, 0, 6)$	7	—326
2 $\text{outfix}(A, 2, 8)$	15	—325.63
3 $\text{outfix}(A, 6, 11)$	26	—325.627000
4 $\text{outfix}(A, 10, 13)$	39	—325.6270000?

В последнем случае выдается сообщение об ошибке.

**procedure** outreal ( $A, N, L$ ); **real**  $A$ ; **integer**  $N, L$ ; ...;

Процедура предназначена для вывода вещественных чисел. Значение параметра  $A$  преобразуется в запись вещественного числа, содержащую порядок. В качестве разделителя мантиссы и порядка используется символ  $E$  (для ЕС ЭВМ) и  $\diamond$  (для БЭСМ-6). Запись вещественного имеет различные виды, которые определяются значением параметра  $N$ . В реализации языка на

ЕС ЭВМ количество знаков в мантиссе не превышает 7, а на БЭСМ-6 — 12.

ЗНАЧЕНИЕ N	ЗАПИСЬ ВЕЩЕСТВЕННОГО
1 $N=0$	$\langle \text{знак} \rangle E \pm \langle \text{порядок} \rangle$
2 $N=1$	$\langle \text{знак} \rangle \langle \text{цифра} \rangle E \pm \langle \text{порядок} \rangle$
3 $N>1$	$\langle \text{знак} \rangle \langle \text{цифра} \rangle . \langle N-1 \text{ цифр} \rangle E \pm \langle \text{порядок} \rangle$

**Пример 3.28.** Пусть вещественной переменной A присвоено значение —365.6279. Используя операторы outreal (A, N, L) с различными значениями N и L, получим следующие записи вещественного:

ОПЕРАТОР	image. pos	ЗАПИСЬ ВЕЩЕСТВЕННОГО
1 outreal (A, 0, 6)	7	—E+02
2 outreal (A, 1, 7)	14	—3E+02
3 outreal (A, 2, 8)	22	—3.6E+02
4 outreal (A, 5, 12)	34	—3.6562E+02
5 outreal(A, 10, 16)	50	—3.6562790E+02
6 outreal(A, 2, 5)	55	—3.6?

В последнем случае выдается сообщение об ошибке.

**procedure** outchar; ...;

Для вывода одной литеры используется процедура outchar (C). Эта процедура помещает литеру, определяемую ее аргументом (C), на текущую позицию буфера (место, указываемое указателем позиции). После выполнения процедуры, указатель позиции увеличивается на 1.

**3.3.6. Закрытие файла.** После завершения работы с файлом его необходимо закрыть. Для этого используется процедура close. Файлы чтения с перфокарт (sysin) и файл вывода на печать (sysout) закрываются в системном классе basicio.

Схема процедуры close имеет следующий вид:

```
procedure close;
  begin if not ОТКРЫТ then ОШИБКА;
    ОТКРЫТ:=false;
    . . .
    image:=notext;
  end close;
```

При вызове процедуры close в начале проверяется, открыт файл или нет. Если файл закрыт, то возникает ошибка. В противном случае устанавливается признак закрытого файла. При закрытии файла вывода, если указатель позиции в image отличен от 1, то текстовое значение, содержащееся в буфере image, передается во внешний файл. В конце процедуры переменной

image присваивается значение `notext`. Попытка чтения (записи) данных из закрытого файла приведет к ошибке.

**3.3.7. Управление печатающим устройством.** В языке симула-67 для вывода результатов работы программы на системное устройство печати определен подкласс `printfile` класса `outfile`. В этом классе имеются несколько процедур, которые позволяют управлять печатающим устройством. Например, с их помощью можно установить длину печатной строки, выводить результаты работы по страницам, производить выдачу на печатающее устройство незаполненной страницы, пропускать несколько пустых строк между двумя последовательными выдачами.

В реализации системного файла вывода, с которым связан объект `sysout` класса `printfile`, длине строки печатающего устройства (АЦПУ) присвоено значение, равное 127. Это значение присваивается переменной `linelength`, недоступной пользователю, и изменить это значение в процессе работы программы нельзя. Длина печатной строки равна размеру буфера вывода (`image`).

Пользователь может определить свой файл вывода на печатающее устройство. В качестве длины строки будет взята длина буфера вывода, определяемая в момент открытия файла. В реализации на ЕС ЭВМ введено ограничение на длину печатной строки: она не должна превышать 127 символов, причем первый символ в буфере является управляющим, и в размер строки не входит. При попытке задания длины печатной строки, большей 127 символов, будет выдано сообщение о превышении допустимой длины строки, и в качестве длины возьмется значение 127. Таким образом, на печать будут выводиться только 127 литер буфера вывода, начиная со второй литеры.

Рассмотрим более подробно процедуры, позволяющие управлять устройством печати.

Процедура `lines per page` предназначена для задания размера страницы. В начале работы симула-программы значение параметра  $N = 71$ , т. е. число печатаемых строк на странице равно 71. Между страницами осуществляется пропуск одной строки. Вызов процедуры `lines per page (M)` влечет за собой изменение размера страницы, т. е. страница будет состоять из  $M$  печатных строк. Если размер страницы изменяется и текущая страница оказывается незаполненной до конца, то оставшиеся на странице строки заполняются пробелами и производится переход к началу следующей страницы, которая будет иметь размер  $M$  строк. Если значение параметра  $M < 1$ , то будет выдано сообщение об ошибке, и размер страницы останется прежним.

Процедура `spacing` позволяет изменять количество строк, пропускаемых между последовательными операциями вывода на печатающее устройство. В начале работы программы выводимые образы буфера вывода печатаются в последовательных строках. Вызов процедуры `spacing(N)` приведет к тому, что между последовательно печатаемыми строками будут размещаться  $N-1$  пустых строк. Эти строки входят в размер страницы. Режим пропуска строк устанавливается в момент выполнения процедуры, а отражение на печати этого режима скажется после следующего (явного или неявного) вызова процедуры `outimage`. Если значение параметра  $N = 0$ , то произойдет наложение печати — буфер вывода будет печататься в одну и ту же физическую строку до тех пор, пока не будет выполнена процедура `spacing(N)` с параметром  $N$ , большим 0. Такая возможность может быть использована, например, для операции подчеркивания ключевых слов. В реализации на ЕС ЭВМ на значения  $N$  наложены ограничения. Если  $N < 0$ , то за значение  $N$  принимается 1, в случае  $N \geq 3$  — значение 3.

Значением процедуры-функции `line` является порядковый номер строки текущей страницы, подлежащей выдаче на печать. В начале работы программы значение, доставляемое этой процедурой, равно 1. После каждого вызова процедуры `outimage` номер строки увеличивается на текущий интервал, который был ранее установлен процедурой `spacing`.

Процедура `eject(N)` служит для установки печатающего устройства на строку, определяемую значением параметра  $N$ , т. е. осуществляется прогон бумаги до заданной строки страницы, что заменяет многократный вызов процедуры `outimage`. После выполнения процедуры бумага на печатающем устройстве будет установлена на строку  $N$  текущей страницы, если  $N \geq \text{line}$ , или на строку  $N$  следующей страницы, если  $N < \text{line}$ . Если значение параметра  $N$  меньше 1 или больше размера страницы, то будет выдано сообщение об ошибке, а действие процедуры будет эквивалентно выполнению пустого оператора.

Процедура `outimage` действует так же, как и процедура `outimage`, определенная в классе `outfile` (см. 3.3.5), но, кроме того, увеличивает номер строки на текущий интервал и производит переход к началу следующей страницы, если текущая страница оказывается уже заполненной.

## ГЛАВА 4

# РАБОТА С СИМУЛА-ПРОГРАММАМИ НА БЭСМ-6 И ЕС ЭВМ

Глава является руководством по практическому применению трансляторов с языка симула-67 для ЭВМ БЭСМ-6 [1] и ЕС ЭВМ [2, 7], которые разработаны группой сотрудников ИПМ им. М. В. Келдыша АН СССР и МИФИ.

В §§ 4.1 и 4.2 приводятся необходимые сведения о трансляторах и правилах их эксплуатации, рассматривается входной язык трансляторов, описываются правила подготовки программ и данных.

Параграфы 4.3 и 4.4 посвящены реализованным в трансляторах средствам отдельной компиляции симула-программ и организации связи с другими системами программирования. Средства пакетной и диалоговой отладки симула-программ рассматриваются в §§ 4.5 и 4.6.

## 4.1. Трансляторы с языка симула-67 для БЭСМ-6 и ЕС ЭВМ

4.1.1. Общие сведения. Трансляторы с языка симула-67 для БЭСМ-6 и ЕС ЭВМ реализованы в виде программных комплексов, состоящих из двух основных частей: компилятора и интерпретирующей системы (ИС). Компиляторы выполняют перевод (компиляцию) исходной симула-программы в программу на языке машины, которую обычно называют объектной программой.

В процессе трансляции оба компилятора выдают листинг исходной симула-программы как в виде покартонной распечатки, так и в форме отредактированного текста, в котором выделены блоки, описания процедур, декларации классов. Отредактированный текст симула-программы содержит двумерную нумерацию, состоящую из номера блока (составного оператора,

процедуры, класса) и номера оператора внутри этого блока, причем на каждой строке печатается только номер первого из присутствующих в нем операторов. По желанию пользователя любая из форм печати исходной программы (или обе сразу) может быть отменена.

Сообщения о синтаксических ошибках, допущенных в симула-программе, либо вставляются непосредственно в текст отредактированной распечатки, либо выдаются после текста исходной программы с указанием двумерного номера оператора, содержащего ошибку. Полный перечень сообщений компиляторов приведен в Приложении 4.

Следует иметь в виду, что некоторые ошибки могут вызывать «наведенные» ошибки. Например, ошибка в описаниях может вызвать появление ошибок вида ИДЕНТИФИКАТОР НЕ ОПИСАН в тех местах программы, в которых употребляются идентификаторы из этих описаний. С другой стороны, список ошибок, выдаваемых компилятором, может оказаться не полным, например, при наличии ошибки, приведенной выше, нельзя полностью проконтролировать корректность использования имен.

При обнаружении некоторых видов грубых ошибок в исходной программе или при нехватке памяти компилятор прерывает трансляцию и выдает соответствующее сообщение. При этом распечатывается информация о внутреннем состоянии компилятора.

В случае нехватки памяти следует применить отдельную компиляцию (см. § 4.3) или увеличить объем оперативной памяти, доступной компилятору.

Кроме текста исходной симула-программы на листинге трансляции присутствуют также сообщения о номере версии компиляторов, общее количество обнаруженных ошибок, информация о затраченных при компиляции ресурсах ЭВМ.

При реализации компиляторов широко применялся язык рефал [5], ориентированный на программирование сложных символьных преобразований. Наличие на машинах БЭСМ-6 и ЕС ЭВМ реализаций рефала [5, 14], практически полностью совместимых по входному языку, позволило значительно сократить трудоемкость разработки симула-компиляторов, особенно компилятора для ЕС ЭВМ, основу которого составили программы ранее разработанного транслятора с языка симула-67 для БЭСМ-6.

Интерпретирующие системы для обоих трансляторов представляют собой совокупность служебных программ, к которым обращается объектная программа при исполнении языковых конструкций с усложненной семантикой. С помощью программ ИС выполняются, например, планирующие операторы, дей-



ствия по созданию объектов, операции ввода-вывода, вычисляются встроенные функции и т. д. Программы ИС хранятся в виде загрузочных модулей и подключаются к объектной программе во время редактирования связей или при ее загрузке в память перед началом счета. Объем подключаемых программ зависит от того, насколько полно используются в исходной программе средства языка симула-67, и колеблется в пределах от 20 до 40 Кбайт для ЕС ЭВМ и от 4 до 8 тыс. слов для БЭСМ-6.

В зависимости от режима, задаваемого при компиляции симула-программы, соответствующая ей объектная программа создается компилятором в одном из двух вариантов: отладочном или счетном.

Отладочные варианты объектных программ обычно применяются для поиска семантических ошибок с использованием средств пакетной и диалоговой отладки, описанных в §§ 4.5 и 4.6. Кроме того, в отладочных программах производится проверка корректности употребления индексных выражений и дистанционных идентификаторов, что значительно облегчает выявление допущенных ошибок. Информация о любой динамической ошибке распечатывается с указанием номера оператора исходной симула-программы, при исполнении которого была обнаружена эта ошибка. Тексты сообщений, выдаваемые программами ИС, с указанием возможных причин ошибок приведены в Приложении 3. В отладочных программах предусмотрено продолжение счета при некоторых видах ошибок.

Счетные объектные программы рекомендуется использовать для выполнения отлаженных алгоритмов: они требуют меньше памяти и времени для исполнения за счет более эффективной компиляции переменных с индексами и дистанционных идентификаторов, но в случае возникновения динамической ошибки счет программы прекращается, а место ошибки указывается с помощью адреса.

Участок оперативной памяти, выделяемый задаче, делится при работе объектной программы на 2 основные части: поле программ и поле данных (ПД).

Поле программ заполняется перед началом счета объектной программы. В нем размещается объектная программа, необходимые ей программы ИС, служебные программы операционной системы, используемые в ИС, а также общий блок, в котором хранятся служебные переменные ИС.

Поле данных на БЭСМ-6 располагается на участке доступной задаче оперативной памяти, оставшейся после заполнения поля программ. На ЕС ЭВМ начальный размер ПД указывается

в поле PARM, передаваемом объектной программе при ее запуске на счет (см. 4.1.3).

Первоначально все поле данных свободно. В процессе работы программы ПД используется для динамического отведения памяти под данные, генерируемые объектной программой. Освобождение памяти, которая занята ставшими недоступными объектами, производится при полном заполнении ПД с помощью программы «сборки мусора». В ПД размещаются переменные, массивы, объекты, тексты и некоторая служебная информация, необходимая для организации работы объектной программы.

Размер памяти, используемый симула-программой в поле данных, можно оценить по следующим формулам:

$$P \approx M + \sum_{k=1}^N P_k (A_k + 5) + T/6 \quad (\text{для БЭСМ-6}),$$

$$P \approx M_B + 4 \left[ M_O + \sum_{k=1}^N P_k (A_k + 14) \right] + T \quad (\text{для ЕС ЭВМ}),$$

где  $P$  — размер памяти в словах (для БЭСМ-6) или в байтах (для ЕС ЭВМ);

$M$ ,  $M_B$ ,  $M_O$  — суммарное количество элементов массивов различных типов, одновременно присутствующих в памяти:  $M_B$  — для массивов типа `boolean` и `character`,  $M_O$  — для массивов остальных типов,  $M$  — для всех массивов ( $M = M_B + M_O$ );

$N$  — количество объектов различных классов, присутствующих в симула-программе;

$P_k$  — максимальное количество объектов  $k$ -го класса, одновременно присутствующих при работе программы;

$A_k$  — количество атрибутов объектов  $k$ -го класса;

$T$  — суммарная длина текстов, обрабатываемых в одновременно присутствующих в ПД блоках, объектах, процедурах.

Перед началом счета объектной программы выдается сообщение о месте размещения ПД в оперативной памяти, а по окончании счета — информация о степени использования выделенной под ПД памяти и числе «сборок мусора», если они производились.

При нехватке памяти в ПД, т. е. при невозможности разместить новый объект (экземпляр блока, процедуры, массива, текста) даже после проведения «сборки мусора», ИС выдает сообщение ИСЧЕРПАНА ПАМЯТЬ В ПОЛЕ ДАННЫХ и счет программы прекращается. В этом случае следует увеличить размер памяти, выделяемой под поле данных (см. 4.1.2, 4.1.3), или обеспечить более экономное использование памяти симула-программой путем, например, размещения в параллельных блоках

массивов, которые не используются одновременно, своевременного уничтожения ссылок на ставшие ненужными объекты, сокращения общего количества сосуществующих объектов и т. д.

4.1.2. Эксплуатация симула-транслятора для БЭСМ-6. Транслятор с языка симула-67 для БЭСМ-6 работает в рамках мониторной системы Дубна [18]. Он состоит из двух персональных библиотек, содержащих программы компилятора (К-библиотека) и интерпретирующей системы (ИС-библиотека). Используя эти библиотеки, можно включить транслятор в состав математического обеспечения в виде стандартного элемента системной библиотеки так же, как это сделано со штатными трансляторами мониторной системы Дубна (фортран [25], forex [32] и т. д.). Для этого необходимо наличие свободного места на системных ресурсах и определенная работа по реорганизации библиотек транслятора. Симула-транслятор можно эксплуатировать и без включения его в системную библиотеку мониторной системы, а пользуясь непосредственно К-библиотекой и ИС-библиотекой. В примере 4.1 приведены управляющие карты мониторной системы, обеспечивающие запуск компилятора, трансляцию симула-программы и исполнение соответствующей ей объектной программы.

Пример 4.1.

\*NAME СИМУЛА-67

\*PERSO: <К-библиотека> *перепись К-библиотеки  
во временную библиотеку.*

\*CALL FICMEMORY

\*NO LOAD LIST

\*MAIN SIMULA

\*EXECUTE

ИМЯ: <имя программы>;  
<программа на симула-67>

еор

\*PERSO: <ИС-библиотека> *считывание ИС-библиотеки.*

\*READ DRUM *трансляция с автокода мадлен.*

\*MAIN <имя программы> *запуск на счет  
объектной программы.*

\*EXECUTE

<данные>

\*END FILE

З а м е ч а н и я.

1. При работе компилятора используется вся оперативная память, поэтому в паспорте задачи нужно подкладывать карту  
ЛИСТЫ 0 — 37.

2. Результатом работы симула-компилятора является программа на автокоде мадлен [18], записанная на магнитный барабан. Карта \*READ DRUM обеспечивает запуск транслятора с автокода, который помещает объектную программу в виде стандартного массива во временную библиотеку.

Для того чтобы сократить время и ресурсы внешней памяти на МБ, используемые при загрузке компилятора, рекомендуется, используя К-библиотеку и статический загрузчик мониторной системы [18], выполнить один раз загрузку компилятора и затем использовать для его запуска карту \*CALL EXECUTE.

Пакет для получения статически собранного компилятора на ленте 538 в зонах, начиная с нулевой, имеет следующий вид:

Пример 4.2.

```
ЛЕНТ 60(538 — 3П) (В ПАСПОРТ)
ЛИСТЫ 0 — 37 (В ПАСПОРТ)
*NAME СТАТИЧЕСКАЯ ЗАГРУЗКА
*CALL FICMEMORY
*PERSO: <К-библиотека>
*CALL OVERLAY
SIMULA (+SIMUL, +SIMLIS, +SIMR, +СКОКТР)
*END RECORD
*END FILE
```

Статически собранный компилятор занимает 28 зон на МЛ или МД. Ниже рассматриваются примеры пакетов запуска статически загруженного компилятора.

Пакет для запуска «трансляция — счет» имеет вид:

Пример 4.3.

1. Паспорт задачи.
2. \*STANDART LIST  
\*FULL LIST  
\*NO LIST
3. \*CALL FICMEMORY
4. \*NO OPTIMIZATION
5. \*CALL EXECUTE
6. ИМЯ: <имя программы>;
7. <текст программы на языке симула-67>
8. EOP
9. \*PERSO: <ИС-библиотека>
10. \*READ DRUM
11. \*MAIN <имя программы>
12. \*NO LOAD LIST
13. \*EXECUTE
14. <данные программы>
15. Конец пакета.

Опишем назначение служебных карт пакета задачи.

1. Паспорт задачи. Кроме стандартных статей паспорта — шифр, время и т. д., в паспорте задачи должны быть статьи заказа магнитных лент (дисков), на которых размещены статически собранный компилятор и интерпретирующая система (ИС).

2. Карты управления редактированием. Эти карты подготавливаются по общим для всех трансляторов мониторной системы Дубна правилам.

Карта \*STANDART LIST задает печать транслируемого текста в отредактированном виде. Редактирование симула-программ заключается в выделении блочной структуры и нумерации операторов. Нумерация оператора — двойная: лексикографический номер блока и номер оператора. Печатается номер лишь первого оператора в строке. При обнаружении в тексте программ синтаксических ошибок, в соответствующие места текста помещаются сообщения.

Карта \*FULL LIST задает печать транслируемой программы как в отредактированном виде, так и покарточно.

Карта \*NO LIST блокирует печать текста программы.

Карта управления редактированием может отсутствовать, по умолчанию устанавливается режим \*STANDART LIST. После трансляции всегда устанавливается режим \*NO LIST.

3. Карта \*CALL FICMEMORY отдает в использование компилятора всю оперативную память машины.

4. Карта \*NO OPTIMIZATION обеспечивает генерацию отладочной объектной программы. Отсутствие такой карты по умолчанию вызывает генерацию счетной объектной программы.

5. Карта \*CALL EXECUTE вызывает запуск статически загруженного компилятора симула-67.

6. Карта ИМЯ: <имя программы>;. Эта карта служит для идентификации объектной программы (стандартного массива). Именем могут служить идентификаторы, состоящие не более, чем из 6 символов. Карта ИМЯ может отсутствовать. В этом случае программе присваивается имя PROGRAM. В качестве имени программы запрещается использование идентификаторов, начинающихся с букв СД, СИ, SBF, RF, S6. Точный перечень имен, запрещенных для употребления, можно получить, распечатав каталог ИС-библиотеки.

8. Карта eor — признак конца симула-программы.

9. Карта \*PERSO: <ИС-библиотека> — карта чтения библиотеки интерпретирующей системы компилятора во временную библиотеку.

10. Карта \*READ DRUM вызывает запуск ассемблера. Если в исходной программе были обнаружены ошибки, то выполнение заданий пакета прекращается.

11. Карта \*MAIN *<имя программы>*. Указание имени объектной программы для запуска задачи на счет. Эта карта может отсутствовать, если не было задано имя программы картой ИМЯ.

12. Карта \*NO LOAD LIST блокирует печать листа загрузки.

13. Карта \*EXECUTE вызывает загрузку и счет объектной программы.

15. Конец пакета — стандартный для мониторной системы.

Пакет для записи объектной программы в библиотеку пользователя отличается от описанного выше тем, что вместо карт, обозначенных номерами 9, 10, 11, 12, 13, 14, в пакет нужно поместить карты:

\*PERSO: *<библиотека пользователя>*

\*READ DRUM

\*TO PERSO: *<библиотека пользователя>*

Запуск такого пакета дополнит библиотеку пользователя программой в виде стандартного массива под именем, указанным в карте ИМЯ.

Пакет для запуска на счет объектных программ, записанных в библиотеку пользователя, составляется так:

Пример 4.4.

Паспорт задачи

\*PERSO: *<библиотека пользователя>*

\*PERSO: *<ИС-библиотека>*, CONT

\*MAIN *<имя программы>*

\*EXECUTE

*<данные программы>*

Конец пакета

Запуск для автономной трансляции совпадает с пакетом для записи объектной программы. В карте ИМЯ называется имя, под которым в библиотеку пользователя будет помещено тело процедуры (класса).

Запуск трансляции и счета симула-программы, в которой использованы автономно оттранслированные процедуры (классы), отличается от запуска «трансляция — счет» тем, что после карты 9 помещается карта: \*PERSO: *<библиотека пользователя>*, CONT. В библиотеке пользователя должны находиться автономно оттранслированные процедуры (классы).

Естественно, все запуски, в которых имеются обращения к библиотеке пользователя, должны содержать в паспорте статью заказа устройства, на котором размещена библиотека.

Для расширения поля данных при счете объектных программ используется карта \*CALL FICMEMORY, которая помещается перед картой \*MAIN *<имя программы>*.

**4.1.3. Эксплуатация симула-транслятора для ЕС ЭВМ.** Программный комплекс, реализующий язык симула-67 на ЕС ЭВМ, работает под управлением ОС ЕС версии 6.0 или более поздних версий и состоит из двух библиотек загрузочных модулей — SIM.CMP и SIM.IS.

В библиотеке SIM.CMP под именем SIMD003 записана программа компилятора, выполняющая перевод исходной симула-программы в объектный модуль ОС ЕС.

Библиотека SIM.IS содержит систему программ, которые используются при счете симула-программы для интерпретации языковых понятий с усложненной семантикой.

Для размещения указанных библиотек на магнитных дисках типа ЕС 5050 требуются следующие ресурсы: библиотека SIM.CMP — 8 цилиндров, библиотека SIM.IS — 6 цилиндров.

Минимальный объем оперативной памяти, требуемый для работы компилятора — 170 Кбайт. В процессе компиляции может запрашиваться дополнительная память, объем которой пропорционален объему исходной симула-программы.

В случае нехватки памяти нужно запустить задачу в большем разделе, если используется режим MFT, или увеличить значение параметра REGION, если работа ведется в режиме MVT. То же самое следует делать, если шаг компиляции заканчивается аварийно с кодом S80A (нехватка памяти).

Компилятор, в зависимости от режима трансляции, генерирует по исходной симула-программе объектный модуль в одном из двух режимов: счетном или отладочном. Отладочный режим задается с помощью ключевого параметра РЕЖ=ОТЛ в предложении, определяющем имя программы, которое помещается перед исходным текстом симула-программы. Например: ИМЯ : SIM67,РЕЖ=ОТЛ. Если не задан отладочный режим, то объектный модуль генерируется в счетном режиме. Имя программы можно не указывать, тогда объектному модулю присваивается имя PRIVATE.

В процессе трансляции программы компилятор строит таблицу относительных адресов операторов, которая выдается в набор данных с DD-именем SYSPRINT. Для программы, оттранслированной в отладочном режиме, эта таблица присутствует и во время счета программы и используется при выдаче сообщений о динамических ошибках, возникающих во время исполнения программы, для указания номера оператора симула-программы, при исполнении которого возникла ошибка.

В счетном режиме также строится и выдается в набор данных с DD-именем SYSPRINT таблица относительных адресов операторов, но при исполнении программы она отсутствует, поэтому в сообщениях об ошибках вместо номера оператора вы-

даются абсолютный и относительный адреса команды (в 16-ричном виде), при исполнении которой обнаружена ошибка. С помощью относительного адреса и таблицы операторов программы, имя которой имеется в сообщении, нетрудно установить оператор, в котором произошла ошибка. Для этого необходимо в таблице операторов найти оператор с максимальным относительным адресом, не превышающим относительный адрес, который указан в сообщении.

Запуск компилятора осуществляется с помощью процедур SIMCLG («трансляция — редактирование — счет») и SIMCL («трансляция — редактирование»), написанных на языке управления заданиями ОС ЕС. Для удобства эксплуатации компилятора тексты этих процедур можно поместить в библиотеку SYS1.PROCLIB, где записаны каталогизированные процедуры для работы со стандартными трансляторами, входящими в состав ОС ЕС. Если такая запись не произведена, то тексты этих процедур следует помещать во входном потоке.

Текст процедуры SIMCLG:

```
//SIMCLG PROC V=SYSLIB,SIMRG=200K,GORG=120K,
// PARM=40000
//SIMULA EXEC PGM=SIMD003,REGION=&SIMRG
//STEPLIB DD DSN=SIM.CMP,DISP=SHR,UNIT=SYSDA,
// VOL=SER=&V
//SYSCARD DD SYSOUT=A
//SYSRED DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYSIN DD DDNAME=SYSIN
//DISKESD DD DSN=&ESD,UNIT=SYSDA,
// SPACE=(80,(200,100)RLSE),DISP=(NEW,PASS)
//DISKTXT DD DSN=&TXT,UNIT=SYSDA,
// SPACE=(80,(200,100)RLSE),DISP=(NEW,PASS)
//DISKRLD DD DSN=&RLD,UNIT=SYSDA,
// SPACE=(80,(200,100)RLSE),DISP=(NEW,PASS)
//SYSUT1 DD DSN=&SYSUT1,DCB=BLKSIZE=1024,
// UNIT=SYSDA,SPACE=(1024,(100,10)RLSE)
//LKED EXEC PGM=IEWL,PARM=(LET),
// COND=((3,LT,SIMULA))
//SYSLIN DD DSN=&ESD,UNIT=SYSDA,
// DISP=(OLD,DELETE)
// DD DSN=&TXT,UNIT=SYSDA,
// DISP=(OLD,DELETE)
// DD DSN=&RLD,UNIT=SYSDA,
// DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLIB DD DSN=SIM.IS,VOL=SER=&V,
```



```
// UNIT=SYSDA,DISP=SHR
//SYSLMOD DD DSN=&GOSET(SIMD003),UNIT=SYSDA
// DISP=(NEW,PASS),SPACE=(CYL,(1,1,1))
//SYSUT1 DD DSN=&SYSUT1,DCB=BLKSIZE=1024,
// UNIT=SYSDA,SPACE=(1024,(100,10),RLSE)
//SYSPRINT DD SYSOUT=A
//GO EXEC PGM=*.LKED.SYSLMOD,COND=((4,LT,LKED)),
// REGION=&GORC,PARM='&PARM'
//SYSOUT DO SYSOUT=A
//SYSIN DD DDNAME=SYSIN
// PEND
```

Поясним назначение некоторых наборов данных, используемых при компиляции и счете симула-программы.

1. Наборы данных &ESD, &TXT, &RLD с DD-именами DISKESD, DISKTXT, DISKRLD.

Эти временные наборы данных с последовательной организацией используются компилятором для размещения объектного модуля, соответствующего исходной симула-программе. В следующем за компиляцией шаге задания эти наборы данных служат входными для редактора связей, выполняющего формирование загрузочного модуля.

2. Выходной набор данных с DD-именем SYSRED.

В этот набор данных записывается отредактированный текст симула-программы с нумерацией операторов. Эта нумерация используется для привязки сообщений об ошибках, выявляемых компилятором и интерпретирующей системой.

3. Выходной набор данных с DD-именем SYSPRINT.

Этот набор данных используется для выдачи служебных сообщений компилятора и сообщений об ошибках.

4. Наборы данных с DD-именами SYSOUT и SYSIN.

Эти наборы данных на шаге GO (счет симула-программы) соответствуют системным файлам SYSOUT и SYSIN, используемым в симула-программе.

5. Набор данных с DD-именем SYSCARD.

Используется для покарточной распечатки исходной симула-программы.

Предполагается, что библиотеки SIM.CMP и SIM.IS располагаются на диске с именем SYSLIB. Если они расположены на другом диске, то в процедурах SIMCLG и SIMCL и во всех примерах вместо V=SYSLIB следует использовать конструкцию V=<имя диска>.

В примере 4.5 приводится пакет управляющих карт для трансляции и счета симула-программы с использованием процедуры SIMCLG.

#### Пример 4.5.

```
//EXPL1 JOB...
// EXEC SIMCLG
//SIMULA.SYSIN DD *
ИМЯ:М15,РЕЖ=ОТЛ;
    <Текст симула-программы>
еор
/*
//GO.SYSIN      DD *
    <данные>
//
```

Карта ИМЯ:М15,РЕЖ=ОТЛ; перед текстом симула-программы в примере 4.5 задает генерацию отладочной программы. Для генерации счетной программы эту карту нужно заменить на ИМЯ:М15;. В следующих примерах мы уже не будем специально выделять карту ИМЯ.

Для отмены выдачи покарточной или отредактированной распечатки исходной симула-программы в пакет примера 4.5 после карты

```
// EXEC SIMCLG
```

нужно вставить карту

```
//SIMULA.SYSCARD DD DUMMY (отмена покарточного ли-
                           стинга)
```

или карту

```
//SIMULA.SYSRED DD DUMMY (отмена редактированного
                           листинга)
```

Присутствие обеих карт отменит выдачу всех форм листинга.

В тех случаях, когда предполагается многократный запуск симула-программ на счет, можно записать полученную редактором связей программу в постоянный набор данных (личную библиотеку пользователя) и затем эксплуатировать ее, меняя лишь данные и не выполняя шагов компиляции и редактирования связей. Для этого достаточно перед картой

```
//GO.SYSIN DD *
```

в примере 4.5 вставить DD-предложение

```
//LKED.SYSLMOD DD DSN=PERSLIB(MODEL), DISP=SHR,
// VOL=SER=MDUSER,UNIT=SYSDA
```

которое обеспечит запись создаваемого редактором связей на шаге LKED загрузочного модуля в раздел MODEL библиотеки PERSLIB, расположенной на диске MDUSER. При исполнении модифицированного таким образом пакета примера 4.5 будет выполнен и счет объектной программы с данными, указанными

после карты //GO.SYSIN DD \*. Последующий счет программы можно производить с помощью пакета, приведенного в примере 4.6.

Пример 4.6.

```
//EXPL JOB...
//GO EXEC PGM=MODEL,PARM=40000,REGION=100K
//STEPLIB DD DSN=PERSLIB,DISP=SHR,UNIT=SYSDA,
// VOL=SER=MDUSER
//SYSOUT DD SYSOUT=A
//SYSIN DD *
    <входные данные для программы MODEL>
/*
//
```

Если объем памяти, указанный в поле PARM, оказывается недостаточным для ПД, то запрашивается необходимое количество памяти у операционной системы с помощью макрокоманды GETMAIN. Если система не может выделить запрашиваемую память, то выдается сообщение ИСЧЕРПАНА ПАМЯТЬ В ПОЛЕ ДАННЫХ и счет задачи прекращается. В этом случае необходимо повторить запуск симула-программы на счет (трансляцию программы можно не повторять, если она была записана в постоянный набор данных) в большем разделе (при работе в режиме MFT) или с увеличенным значением параметра REGION (в режиме MVT).

По окончании работы объектной программы вырабатывается код условия, передаваемый на общем регистре 15 в вызывающую программу. Код условия равен 0, если управление покидает симула-программу через завершающий ее символ end (нормальное окончание), а если во время исполнения объектной программы были динамические ошибки или возникла нехватка памяти в поле данных, то код условия устанавливается равным 4.

При применении средств отдельной компиляции симула-программ (см. 4.3) отдельные загрузочные модули нужно получать и хранить без подключения необходимых им программ интерпретирующей системы. То же самое можно делать для экономии пространства в библиотеках, содержащих большое количество откомпилированных симула-программ, поскольку в этом случае исключается многократное присутствие одних и тех же программ ИС, используемых в нескольких симула-программах.

Для получения по исходной симула-программе загрузочного модуля без подключения программ ИС удобно пользоваться процедурой SIMCL, текст которой приводится ниже.

```
//SIMCL PROC V=SYSLIB,SIMRG=200K,
// LKEDPRM='NCAL, NOMAP'
//SIMULA EXEC PGM=SIMD003,REGION=&SIMRG
//STEPLIB DD DSN=SIM.SMP,DISP=SHR,UNIT=SYSDA,
// VOL=SER=&V
//SYSCARD DD SYSOUT=A
//SYSRED DD SYSOUT=A
//SYSPRINT DD SYSOUT=A
//SYSIN DD DDNAME=SYSIN
//DISKESD DD DSN=&ESD,UNIT=SYSDA,
// SPACE=(80,(200,100),RLSE,DISP=(NEW,PASS)
//DISKTXT DD DSN=&TXT,UNIT=SYSDA,
// SPACE=(80,(200,100),RLSE),DISP=(NEW,PASS)
//DISKRLD DD DSN=&RLD,UNIT=SYSDA,
// SPACE=(80,(200,100),RLSE),DISP=(NEW,PASS)
//SYSUT1 DD DSN=&SYSUT1,DCB=BLKSIZE=1024,
// UNIT=SYSDA,SPACE=(1024,(100,10),RLSE)
//LKED EXEC PGM=IEWL,PARM='&LKEDPRM',
// COND=((3,LT,SIMULA))
//SYSLIN DD DSN=&ESD,UNIT=SYSDA,
// DISP=(OLD,DELETE)
// DD DSN=&TXT,UNIT=SYSDA,
// DISP=(OLD,DELETE)
// DD DSN=&RLD,UNIT=SYSDA,
// DISP=(OLD,DELETE)
// DD DDNAME=SYSIN
//SYSLIB DD DSN=SIM.IS,VOL=SER=&V,UNIT=SYSDA,
// DISP=SHR
//SYSLMOD DD DSN=&GOSET(SIMD003),UNIT=SYSDA,
// DISP=(NEW,PASS),SPACE=(CYL,(1,1,1))
//SYSUT1 DD DSN=&SYSUT1,DCB=BLKSIZE=1024,
// UNIT=SYSDA,SPACE=(1024,(100,10),RLSE)
//SYSPRINT DD SYSOUT=A
// PEND
```

В примере 4.7 процедура SIMCL используется для записи загрузочного модуля симула-программы в раздел MOD библиотеки PERSLIB.

**Пример 4.7.**

```
//EXPL2 JOB...
// EXEC SIMCL
//SIMULA.SYSIN DD *
    <текст симула-программы>
```

**eof**  
/\*

```
//LKED.SYSLMOD DD DSN=PERSLIB(MOD),DISP=SHR,
// VOL=SER=MDUSER,UNIT=SYSDA
```

Для запуска на счет симула-программы, хранящейся в библиотеке без подключения программы ИС, можно воспользоваться загрузчиком (пример 4.8) или редактором связей [17].

**Пример 4.8.**

```
//EXPL3 JOB...
//GO EXEC PGM=LOADER,PARM='SIZE=100000/40000',
// REGION=100K
//SYSLIN DD DSN=PERSLIB(MOD),DISP=SHR,
// UNIT=SYSDA,VOL=SER=MDUSER
//SYSLIB DD DSN=SIM.IS,DISP=SHR,
// VOL=SER=SYSLIB,UNIT=SYSDA
//SYSLOUT DD SYSOUT=A
//SYSOUT DD SYSOUT=A
//SYSIN DD *
      <данные для программы>
/*
//
```

## 4.2. Входной язык транслятора

Трансляторы с языка симула-67 для БЭСМ-6 и ЕС ЭВМ практически полностью совместимы по входному языку, за основу которого принят эталонный язык симула-67, описанный в работах [10, 26]. В данном разделе мы рассмотрим ограничения, уточнения и расширения эталонного языка, введенные при его реализации, а также правила подготовки программ и данных при работе с трансляторами.

**4.2.1. Ограничения, накладываемые трансляторами.** Во входном языке обоих трансляторов наложены следующие ограничения на эталонный язык симула-67:

- идентификаторы могут иметь произвольную длину, но различаются они только по первым 6 (на БЭСМ-6) или 7 (на ЕС ЭВМ) символам. В состав идентификаторов могут входить русские буквы. Пробелы внутри идентификаторов игнорируются,

- в качестве разделителей параметров процедур допускаются только запятые,

- параметры-массивы передаются только по ссылке,

- запрещен выход из тела процедуры-функции с помощью оператора перехода,

- результат операции возведения в степень считается целым только тогда, когда основание степени — целое число, а показатель — целое положительное число,

— идентификаторы стандартных процедур и процедуры-атрибуты системных классов не могут передаваться в качестве фактических параметров,

— максимально допустимая вложенность блоков равна 15,

— системные классы `simset` и `simulation` можно использовать в качестве префиксов к блоку только на втором лексикографическом уровне программы, например:

```
begin real A, B; ...
```

```
  comment начало первого блока;
```

```
  simulation begin ... end simulation;
```

```
end первого блока;
```

**З а м е ч а н и е.** На ЕС ЭВМ допускается использование идентификаторов `simset` и `simulation` в качестве префиксов к первому блоку программы, т. е. симула-программа может иметь вид

```
simulation begin ... end
```

— процедуры ввода-вывода, определенные в системных классах `file`, `infile`, `outfile`, не являются виртуальными, т. е. не могут быть переопределены пользователем,

— длина текстовой строки в симула-программе не должна превышать 255 символов,

— при реализации переключателей наложены следующие ограничения:

а) на ЕС ЭВМ переключательный список должен состоять только из меток,

б) на БЭСМ-6 значение указателя переключателя не определено, если значение его индексного выражения выходит за границы переключательного списка,

— на БЭСМ-6 значения индексов и границ массивов не должны превышать 32767,

— запрещено переопределение следующих стандартных идентификаторов: `trace`, `entry`, `delete`.

**4.2.2. Уточнения эталонного языка.** Средства ввода-вывода и процедуры случайного выбора, определенные в эталонном языке симула-67, допускают уточнения некоторых параметров и действий в конкретных реализациях. В данном разделе мы рассмотрим такие уточнения, сделанные в процессе разработки трансляторов для БЭСМ-6 и ЕС ЭВМ.

1. В системном классе `sysout` значение переменной `linelength`, определяющей количество литер в строке печатающего устройства, равно 127.

2. Буферы ввода-вывода для системных файлов `sysin` и `sysout` в начале работы программы открыты и заполнены про-

белами. Буфер ввода, для файла определенного пользователем, заполняется первым образом из внешнего файла в момент его открытия процедурой open.

3. Определяемый пользователем внешний файл идентифицируется на ЕС ЭВМ с помощью имени DD-предложения языка управления заданиями ОС ЕС [17], указывающего расположение набора данных, который будет связан с соответствующим объектом класса infile или outfile. На БЭСМ-6 расположение внешнего файла указывается с помощью математического номера устройства ввода-вывода (ленты, диска, магнитного барабана) и начального номера зоны, с которого начинается этот файл.

Ввод-вывод результатов на внешние устройства в трансляторе на БЭСМ-6 реализован с помощью программ бесформатного ввода-вывода языка фортран. Ввиду того, что в фортране определена своя нумерация внешних устройств (логическая), не совпадающая с нумерацией системы (математической), между математическими и логическими номерами введено взаимно однозначное соответствие:

логич.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
матем.	41	42	43	44	45	46	47	50	51	52	53	54	55	56	57

Магнитный барабан — математический номер 60, логический номер 16. Ввод с перфокарт — математический номер 61, логического не имеет. АЦПУ — математический номер 62, логического не имеет. Вывод на перфокарты — математический номер 63, логического не имеет. В случае выдачи ошибок программами бесформатного ввода-вывода, в них указывается логический номер устройства, на котором располагается файл.

Рассмотрим пример организации работы симула-программы с внешним файлом на ЕС ЭВМ. Пусть необходимо открыть файл с именем INPUT и ввести ряд целых чисел с последовательного набора данных, расположенного на диске с именем MD0001, в массив А. После операции ввода необходимо закрыть файл.

**Пример 4.9.**

**ref (infile) ВВОД; text БУФ; integer K;**  
**integer array A[1 : 1000];**

(1) ВВОД:—new infile ('INPUT');

(2) БУФ:—blanks (80);

(3) ВВОД. open (БУФ);

**for K:=1 step 1 until 1000 do**

- (4) A[K]:=BВOD. inint;
- (5) BВOD. close;

. . .

Оператор (1) заносит в ссылочную переменную BВOD ссылку на новый объект класса infile, который идентифицируется именем INPUT. Это имя указывает то DD-предложение, которое должно быть задано на шаге задания, выполняющего счет симула-программы. В качестве DD-имени можно использовать текстовую переменную, которая в момент открытия файла должна иметь значение, отличное от notext, причем его первые 8 символов должны быть допустимыми для имен DD-предложений языка управления заданиями в ОС ЕС. Например:

```
//INPUT DD DSN=EXPDATA,DISP=SHR,VOL=SER=MD0001,  
// UNIT=SYSDA
```

Предполагается, что набор данных EXPDATA состоит из 80-байтовых записей (образов).

Оператор (2) заносит в текстовую переменную БУФ ссылку на текст длиной 80 символов.

Оператор (3) открывает доступ к файлу, определенному в операторе (1), задавая одновременно его буфер (БУФ) и обеспечивая ввод первой записи.

Оператор (4) осуществляет ввод целых чисел с внешнего файла в массив.

Оператор (5) закрывает файл с именем INPUT.

Файл вывода на внешнее устройство организуется так же, как и файл ввода.

При реализации обменов с внешними устройствами в трансляторе на ЕС ЭВМ используется последовательный метод доступа QSAM. При этом обработка записей производится в режиме пересылки [29]. Записи файлов должны всегда иметь фиксированную длину, равную длине текстового значения image.

Для определения файлов (наборов данных) используются DD-предложения, в которых необходимо указать параметр DCB. В качестве операндов параметра DCB должны присутствовать:

1. Операнд RECFM — задает формат записи. Допускаются заблокированные или несблокированные записи (F или FB).

2. Операнд LRECL — длина записи равная длине текстового значения image,

3. Операнд BLKSIZE — размер блока, равный длине записи или кратный ей, если записи заблокированы.

Для того чтобы выполнить на БЭСМ-6 действия, заданные в примере 4.9, в приведенном фрагменте симула-программы при задании файла с именем 'INPUT' в операторе (1) необходимо указать математический номер устройства и номер зоны (в



десятичной системе), с которой располагаются вводимые данные. Пусть, например, данные записаны на ленте (диске) с номером 2095, начиная с зоны 48. Тогда участок симула-программы из примера 4.9 выполнит ввод целых чисел в массив А, если оператор (1) записать в виде

ВВОД:—new infile ('INPUT', 55, 48);

и поместить в паспорт задачи карту заказа ленты (диска): ЛЕНТ 55(2095).

4. Если размер буфера ввода с внешней памяти не соответствует ранее использованному буферу вывода на внешнюю память, возникает ошибка ввода-вывода,

5. При реализации вероятностных процедур языка использовался датчик псевдослучайных чисел, равномерно распределенных на интервале (0, 1), определенный в языке симула-67.

Это распределение дает процедура главной выборки `psrand (A)`. При каждом обращении к процедуре `psrand` изменяется значение параметра А по формуле  $A_{i+1} = \text{RES}((A_i + 5^{2p+1}) \div 2^n)$ , а значение `psrand` вычисляется как  $\text{psrand} = A_i \cdot 2^{-n}$ . При этом  $\text{psrand} \in (0, 1)$ . При реализации на ЕС ЭВМ выбраны  $n=30$ ,  $p=6$ , а на БЭСМ-6 —  $n=19$ ,  $p=4$ .

4.2.3. Расширения эталонного языка. В процессе разработки симула-трансляторов для БЭСМ-6 и ЕС ЭВМ в их входной язык были введены следующие дополнения к эталонному языку:

- введен описатель **common** для статического размещения данных целого, вещественного и булевского типа. Правила использования этого описателя даны в 4.4.2;

- определены средства связи с программами, написанными на других языках программирования (см. § 4.4);

- расширены и уточнены средства раздельной компиляции симула-программ (см. § 4.3);

- разработаны средства отладки симула-программ, позволяющие следить за динамикой их выполнения (см. §§ 4.5, 4.6);

- расширены возможности передачи параметров в классы: а) параметры со спецификацией  $\langle \text{тип} \rangle$  могут передаваться по наименованию, б) допускается использование параметров, специфицированных как **procedure** или  $\langle \text{тип} \rangle$  **procedure**.

В этих случаях соответствующие фактические параметры должны быть определены в блоках, охватывающих декларации классов, которые имеют формальные параметры, упомянутые в пп. а) и б).

В качестве примера использования параметров-процедур и вызова по наименованию в декларациях классов рассмотрим

описание класса объектов, каждый из которых во время работы имитационной модели периодически проверяет истинность некоторого логического выражения и вызывает соответствующую процедуру в том случае, если это выражение имеет значение **false**. Допустим, что перед вызовом процедуры нужно распечатать текст, идентифицирующий выражение со значением **false**. Тогда соответствующая декларация класса будет иметь следующий вид:

Пример 4.10.

```
process class КОНТРОЛЬ (В, Р, ИМЯ, ПЕРИОД):  
  name В; boolean В; procedure Р; text ИМЯ; real ПЕРИОД;  
  begin РАБОТА: if not В then begin outtext ('ИМЯ'): Р end;  
    hold (ПЕРИОД); goto РАБОТА;  
end КОНТРОЛЬ;
```

Чтобы воспользоваться приведенной декларацией класса для контроля за выполнением некоторых условий нужно создать и запустить в работу объекты класса КОНТРОЛЬ. Например, если каждые 5 единиц системного времени надо проверять выполнение условия  $X > 10$  и вызывать процедуру КОРРХ, если оно не выполняется, а каждые 15 единиц времени контролировать условие  $Y > \sin(X)$  и обращаться к процедуре КОРРУ при его невыполнении, то достаточно употребить операторы

```
activate new КОНТРОЛЬ (X > 10, КОРРХ, 'X НЕ БОЛЬШЕ  
10', 5);  
activate new КОНТРОЛЬ (Y > SIN(X), КОРРУ, 'Y >  
> SIN(X) — НАРУШЕНО', 15);
```

Предполагается что X, Y, КОРРХ, КОРРУ описаны в блоках, охватывающих декларацию класса КОНТРОЛЬ. Дальнейшие параметры использования вызова по наименованию параметров деклараций классов можно найти в разделе 5.2, в котором описывается применение языка симула-67 для моделирования непрерывно-дискретных систем.

4.2.4. Подготовка программ и данных. Подготовка (перфорация) симула-программ производится с учетом кодировки основных символов и операций языка, приведенной в Приложении 5.

При подготовке перфокарт необходимо учитывать следующее:

- компилятор воспринимает только первые 72 символа карты,

- нельзя переносить на следующую карту основные символы языка (**begin**, **if** и т. д.),

— пробелы считаются незначащими символами в любом месте программы, кроме строк, основных символов и знаков операций,

— допускается 2 способа кодировки символа присваивания ссылок (:—) эталонного языка: :— и %— (двоеточие минус и процент минус),

— ограничителями строк являются два подряд стоящих апострофа: (") ,

— литеры (константы типа `character`) обрамляются символами " (двойная кавычка),

— знак десятичного порядка в тексте симула-программы кодируется символом  $\propto$ . На ЕС ЭВМ допускается использование буквы E в роли десятичного порядка, если в записи числа ему предшествует цифра.

**З а м е ч а н и е.** На некоторых устройствах подготовки данных и печатающих устройствах для ЭВМ БЭСМ-6, символ  $\propto$  изображается как  $\diamond$ .

Данные, предназначенные для ввода симула-программой через системное устройство ввода (ему соответствует файл `sysin`, открываемый перед началом работы симула-программы пользователя), могут размещаться в позициях 1—80 вводимых перфокарт или их образов, хранящихся на внешней памяти.

### 4.3. Раздельная компиляция симула-программ

Применение средств раздельной компиляции позволяет избежать в процессе отладки многократной перетрансляции уже отлаженных частей программ, а также создавать проблемно-ориентированные библиотеки классов и процедур, которые могут служить основой для разработки пакетов прикладных программ для широкого спектра предметных областей.

Вместе с аппаратом иерархий деклараций классов, составляющим основу языка симула-67, средства раздельной компиляции дают возможность наглядно отображать и фиксировать в архивах вычислительных систем иерархические структуры понятий, постепенно детализировать описания сложных объектов и систем в процессе их исследования или проектирования.

В п. 4.3.1 рассматривается создание библиотек классов и процедур, в п.4.3.2 — использование библиотечных классов и процедур.

**4.3.1. Создание библиотек классов и процедур.** Оператор `entry`. В системах программирования на базе языка симула-67, реализованных на ЭВМ БЭСМ-6 [1] и ЕС ЭВМ [7], библиотеки классов и процедур состоят из двух частей:

1. Программные модули, полученные в результате трансляции исходных программ на языке симула-67. На БЭСМ-6 эту часть составляют общие или персональные библиотеки программ в виде стандартных массивов мониторной системы Дубна [18], а на ЕС ЭВМ — библиотеки загрузочных модулей [16].

2. Информация о библиотечных классах и процедурах, необходимая симула-компилятору при трансляции программ, использующих эту библиотеку. Эта информация содержит, в основном, сведения об атрибутах классов и параметрах процедур, хранящихся в библиотеке.

Таким образом, для создания библиотеки классов и процедур необходимо обеспечить сохранение в постоянных наборах данных в архивах вычислительной системы обеих указанных составных частей.

Сохранение программных модулей выполняется с помощью средств организации библиотек программ, имеющихся в составе математического обеспечения ЭВМ. Ниже будут приведены примеры применения этих средств при организации библиотек классов и процедур.

Запись необходимой информации о процедурах и классах, входящих в состав библиотеки, выполняет компилятор с языка симула-67 при трансляции симула-программ, содержащих их описания. Для указания имен процедур и классов, информацию о которых нужно сохранить для дальнейшего использования, а также для задания наборов данных, в которые нужно записать эту информацию, используется оператор `entry`. Он имеет следующий синтаксис:

```
entry (<список имен>, (<указатель набора данных>)),  
где <список имен>::= $\langle$ элемент списка имен $\rangle$  | <список имен>,  
    <элемент списка имен>  
<элемент списка имен>::= $\langle$ имя $\rangle$  | <имя>= $\langle$ внешнее имя $\rangle$   
<имя>::= $\langle$ идентификатор класса $\rangle$  | <идентификатор процеду-  
ры> | <идентификатор атрибута>  
<внешнее имя>::= $\langle$ идентификатор $\rangle$ 
```

В конструкции <список имен> перечисляются идентификаторы тех процедур и классов, определенных в данной программе, информацию о которых компилятор должен сохранить в указанном наборе данных. Если для некоторого имени (идентификатора процедуры, класса, атрибута) в конструкции <элемент списка имен> задано <внешнее имя>, то при записи информации оно заменяет исходный (внутренний) идентификатор, обозначающий процедуру (класс, атрибут) в транслируемой программе. Это означает, что в создаваемой библиотеке соответствующая процедура (класс, атрибут класса) будет представ-

лена не под своим идентификатором, а под внешним именем. При переименовании атрибутов классов следует учитывать, что новое имя получают и одноименные атрибуты других классов, информация о которых записывается в один набор данных.

В программном модуле, создаваемом компилятором по исходной симула-программе, будут присутствовать входные точки для процедур и классов, упомянутых в операторе `entry`. Имя входной точки будет совпадать с внешним именем, если оно задано в соответствующем элементе списка имен, а если внешнее имя не указано, то входная точка получит имя, совпадающее с идентификатором процедуры (класса).

Конструкция `<указатель набора данных>` имеет различный синтаксис и семантику во входных языках симула-компиляторов для БЭСМ-6 и ЕС ЭВМ.

При трансляции симула-программ на БЭСМ-6 в роли указателя набора данных (УКН) должно использоваться пятизначное целое число, первые две цифры которого трактуются как математический номер устройства, а остальные три цифры — как номер зоны (в восьмеричной системе счисления) на этом устройстве (магнитной ленте или диске), начиная с которой компилятор будет располагать информацию о классах и процедурах, упомянутых в списке имен оператора `entry`.

На ЕС ЭВМ УКН задает имя DD-предложения, определяющего набор данных, в который симула-компилятор будет записывать информацию о классах и процедурах, входящих в состав библиотеки. Синтаксически УКН является в данном случае идентификатором, в состав которого не могут входить русские буквы, не совпадающие по начертанию с латинскими, поскольку они недопустимы в DD-именах языка управления заданиями в ОС ЕС [17].

Объем записываемой компилятором информации пропорционален суммарному числу идентификаторов процедур, классов, их атрибутов и формальных параметров и не превышает 25 байтов на один идентификатор. При записи информации компилятор производит ее печать.

На БЭСМ-6 запись информации производится в символьном виде в формате, принятом для текстовых файлов мониторной системы Дубна, а на ЕС ЭВМ — во внутреннем коде компилятора. При работе на ЕС ЭВМ следует учитывать, что симула-компилятор записывает указанную информацию в виде последовательности несблокированных записей фиксированной длины, используя метод доступа BSAM, причем для размещения информации об одной процедуре (декларации класса) расходуется как минимум, одна новая запись. Длина записи берется из DD-предложения, указанного в операторе `entry`, или метки

набора данных. В связи с этим, для экономного расходования внешней памяти рекомендуется использовать для хранения информации последовательные наборы данных с небольшой длиной записи (80—1024 байт).

Оператор `entry` должен быть употреблен в том же блоке, где описаны все упомянутые в нем процедуры и классы. Если в одной программе имеется несколько операторов `entry`, в которых указан один и тот же набор данных, то в него будет записана информация о процедурах и классах, упомянутых во всех таких операторах.

Рассмотрим пример применения оператора `entry`. Пусть имеются классы `A`, `B`, `C` и процедуры `P`, `Q`, `F`, которые нужно оттранслировать, обеспечив возможность их использования из других программ. Это можно сделать с помощью следующей программы:

**Пример 4.11.**

ИМЯ:NEWLIB;

**begin**

**class** `A` (`X`); **real** `X`; *<тело класса A>*;

**A class** `B` (`Y`); **integer** `Y`; *<тело класса B>*;

**class** `C` (`M`, `N`, `P`); **text** `M`, `N`, `P`;

**begin**

**procedure** `P` (`Z`); **ref** (`A`) `Z`; *<тело P>*;

**real procedure** `Q`; *<тело Q>*;

**entry** (`P`, `Q`, (`УКН`));

*<операторы тела класса C>*

**end C**;

**procedure** `F` (`X`, `Y`); **integer** `X`, `Y`; *<тело F>*;

**entry** (`A`=`EXTA`, `X`=`AX`, `B`, `C`, `F`, (`УКН`));

**end**

Через `УКН` в примере 4.11 обозначен указатель набора данных, куда при трансляции программы компилятор будет записывать информацию о процедурах `P`, `Q`, `F`, классах `A`, `B`, `C` и их формальных параметрах и атрибутах.

Наличие во втором операторе `entry` конструкций `A=EXTA` и `X=AX` обеспечивает при записи информации переименование класса `A` на `EXTA`, а его атрибута `X` на `AX`.

При трансляции программы примера 4.11 на БЭСМ-6 в качестве `УКН` следует указать в виде пятизначного числа номер устройства и номер зоны, начиная с которой будет записана информация, и обеспечить запись полученного после трансляции стандартного массива в персональную библиотеку. Это можно сделать с помощью следующего пакета управляющих карт мониторной системы Дубна.

### Пример 4.12.

\*NAME ПРИМЕР 4.11 на БЭСМ-6

\*CALL EXECUTE *запуск симула-компилятора*

ИМЯ: NEWLIB;

```
begin . . . . .
    entry (P, Q, (55200));
    . . . . .
    entry (A=EXTA, X=AX, B, C, F, (55200));
end
eop
```

\*PERSO:55300

*запись программ*

\*READ DRUM

*в персональную*

\*TO PERSO:55300, 20, A, B, C, P, Q, F

*библиотеку*

\*END FILE

В паспорте задачи должно быть заказано устройство с номером 60, где располагается статически собранный компилятор симула-67 и устройство с номером 55, где будет расположена создаваемая библиотека.

После выполнения пакета из примера 4.12 на устройстве 55 с зоны 200 будет расположена информация о классах A, B, C и их атрибутах (класс A будет представлен под именем EXTA) и процедурах P, Q, F, а в персональной библиотеке, начинающейся с зоны 300 того же устройства, появится программа (стандартный массив) с именем NEWLIB, имеющая входные точки EXTA, B, C, P, Q, F.

Трансляцию программы из примера 4.11 на ЕС ЭВМ можно выполнить с помощью следующего пакета:

Пример 4.13.

// EXEC SIMCL

//SIMULA.SYSIN DD \*

ИМЯ: NEWLIB;

```
begin . . . . .
    entry (P, Q, (SIMLIB));
    . . . . .
    entry (A=EXTA, X=AX, B, C, F, (SIMLIB));
end
eop
```

/\*

//SIMLIB DD DSN=CLASSLIB(INFLIB),DISP=(NEW,KEEP),

// UNIT=SYSDA,SPACE=(1024,(30,20,10)),

// DCB=BLKSIZE=1024,VOL=SER=USERTOM

//LKED.SYSLMOD DD DSN=CLASSMOD(NEWLIB),DISP=SHR,

// VOL=SER=USERTOM,UNIT=SYSDA

При выполнении данного пакета на диске USERTOM будет создана новая библиотека с именем CLASSLIB и в ее раздел

INFLIB симула-компилятором будет записана информация о классах A, B, C и процедурах P, Q, F. В раздел NEWLIB библиотеки CLASSMOD редактор связей поместит загрузочный модуль (без подключения программ интерпретирующей системы), соответствующий программе из примера 4.11. Этот загрузочный модуль будет иметь имя секции NEWLIB, заданное в карте ИМЯ: NEWLIB; и входные точки EXTA, B, C, P, Q, F, заданные в операторах entry.

**З а м е ч а н и е.** В операторе entry может отсутствовать конструкция <указатель набора данных>. В этом случае компилятор не сохраняет информацию об упомянутых в таком операторе процедурах и классах, но создает для них в соответствующем программном модуле входные точки, что позволяет в дальнейшем использовать эти процедуры и классы из других программ. При этом пользователь должен, используя такие процедуры, указать их заголовки со спецификациями параметров и вместо тела задать внешнее имя посредством оператора code (<внешнее имя>), а при использовании классов задать их декларации в виде

```
class <внутреннее имя> (<параметры>);
    <спецификации параметров>;
begin <описания атрибутов>;
    code (<внешнее имя>)
end;
```

Этот прием можно использовать при автономной компиляции деклараций классов, ссылающихся друг на друга посредством генераторов объектов, дистанционных идентификаторов или присоединяющих операторов.

Приведем пример использования операторов entry и code для раздельной трансляции деклараций классов и процедур, обращающихся друг к другу и к глобальным переменным.

Пусть имеется следующая симула-программа, которую будем называть основной.

**П р и м е р 4.14.**

```
begin integer I, J; real K;
simulation begin ref (E) EE; ref (A) RA; ref (EP) EEP;
class E(X, Y); integer X, Y;
    begin
        outtext ('ИЗ E [X+Y]:'); outint (X+Y, 5);
    end;
process class EP(Z); integer Z;
    begin integer ZZ;
        procedure P(X); integer X;
    begin
```



```

outtext ('P(X); X, Z, I:');
  outint (X, 5); outint (Z, 5); outint (I, 5);
end P(X);
  ZZ:=Z+J; outtext ('ИЗ EP (ZZ=Z+J):'); outint (ZZ, 5);
passivate; outtext ('PASSIV.:'); outint (ZZ+I, 5);
end EP;
class A(M); integer M;
  begin ref (E); RE; RE:—EE;
outtext ('ИЗ A (M)'); outint (I+J+M, 8); outint (RE.X, 8);
  end A;
EE:—new E(10, 20); EEP:—new EP(149); RA:—new A(100);
activate EEP; outtext ('EXTERNAL:');
outint (EE.X+EEP.Z+EEP.ZZ, 5); EEP.P (EE.X); EEP.P(I);
inspect EE when E do
  outint(X+Y, 5);
  activate EEP; outimage;
end; end

```

Для того чтобы скомпилировать классы E, EP и процедуру P автономно от основной программы, нужно составить следующую программу, содержащую их описания, программное окружение (строки 1,2,3,N,N+1) и оператор entry, задающий для них внешние имена:

Пример 4.15.

```

1   begin integer I, J; real K;
2   simulation begin ref (E) EE; ref (A) RA; ref (EP) EEP;
3   class A(M); integer M;
    begin ref (E) RE; code (A); end A;
    class E(X, Y); integer X, Y;
      begin
        outtext ('ИЗ E [X+Y]:'); outint (X+Y, 5);
      end;
process class EP(Z); integer Z;
  begin integer ZZ
procedure P(X); integer X;
  begin
    outtext ('P(X): X, Z, I:');
    outint (X, 5); outint (Z, 5); outint (I, 5);
  end P(X);
  ZZ:=Z+J; outtext ('ИЗ EP (ZZ=Z+J):'); outint (ZZ, 5);
passivate; outtext ('PASSIV.:'); outint (ZZ+I, 5);
  end EP;
N — 1 entry (E, EP, P);
N   end;
N+1 end

```

После того как будет выполнена трансляция программы из примера 4.15, можно сократить основную программу, заменив в ней тела классов E, EP и процедуры P ссылками на соответствующие внешние программы:

**Пример 4.16.**

```
begin integer I, J; real K;
simulation begin ref (E) EE; ref (A) RA; ref (EP) EEP;
  class E(X, Y); integer X, Y;
    code (E);
  process class EP(Z); integer Z;
    begin integer ZZ;
    procedure P(X); integer X; code (P);
    code (EP);
    end EP;
  class A(M); integer M;
    begin ref (E) RE; RE:—EE;
  outtext ('ИЗ A (M)'); outint (I+J+M, 8); outint (RE.X, 8);
    end A;
  EE:— new E(10, 20); EEP:—new EP(149); RA:— new A(100);
  activate EEP; outtext ('EXTERNAL:');
  outint (EE.X+EEP.Z+EEP.ZZ, 5); EEP.P(EE.X); EEP.P(I);
  inspect EE when E do
    outint (X+Y, 5);
  activate EEP; outimage; entry (A);
end;
end
```

Отметим, однако, что более удобным способом задания программного окружения для процедур и классов с глобальными переменными является описание некоторого объемлющего класса, содержащего в качестве атрибутов эти процедуры и классы, а также необходимые им глобальные величины. Примеры таких классов приведены в главе 5.

**4.3.2. Использование библиотек автономно скомпилированных процедур и классов. Описатель external.** Для использования классов и процедур, записанных в некоторой библиотеке, необходимо указать симула-компилятору местонахождение набора данных, в котором записана информация об этих процедурах и классах. Это делается с помощью специального описания, начинающегося с описателя **external** и имеющего следующий синтаксис:

```
external <список переименований> (<указатель набора дан-
ных>);
<список переименований>::=<пусто>|
  <переименование>,<список переименований>
```

⟨переименование⟩::=⟨внутреннее имя⟩=⟨внешнее имя⟩

В роли внутреннего и внешнего имен в конструкции ⟨переименование⟩ могут выступать идентификаторы классов и их атрибутов, а также идентификаторы процедур.

Синтаксис и семантика конструкции ⟨указатель набора данных⟩ полностью совпадают с синтаксисом и семантикой аналогичной конструкции в операторе entry.

Встретив описание, начинающееся с символа **external**, симула-компилятор читает информацию из набора данных и преобразует ее во внутренний код компилятора. Программа, содержащая такое описание, может использовать все процедуры и классы, информация о которых была записана в наборе данных. Если в описании заданы переименования, то в процессе ввода информации компилятор заменяет имеющиеся там вхождения внешних имен на внутренние. При этом обращаться к библиотечным процедурам, классам и атрибутам классов в данной программе нужно, конечно, по внутренним именам.

В качестве примера употребления описателя **external** рассмотрим использование библиотечных классов и процедур, определенных в примере 4.11. Если мы хотим обратиться к классу А и его атрибуту Х, представленным в библиотеке под именами EXTA и AX, посредством внутренних имен AA и XX, к классу В — посредством имени BB, а к остальным классам и процедурам по прежним именам, то в первом блоке программы мы должны задать описание, начинающееся с символа **external**, которое должно предшествовать употреблению идентификаторов AX, XX и BB:

Пример 4.17.

```
ИМЯ: USING;  
begin external AA=EXTA, XX=AX, BB=B, (УКН);  
  ref (AA) RA; ref (E) RE; ref (C) RC; integer I, J; real H;  
  C class E; <тело класса E>;  
  BB class BE; <тело класса BE>;  
  RA:—new AA (...); RE:—new E (...), RC:—new C (...);  
  F (I+1, J+8); H:=RA.XX; RC.P (RA); I:=RC.Q;  
  . . .  
end
```

На месте УКН в программе, использующей библиотечные классы и процедуры, нужно указать тот же набор данных, который задавался при создании библиотеки.

Опишем пакеты для трансляции и счета программы из примера 4.17 на БЭСМ-6 и ЕС ЭВМ. Предполагается, что библиотечные классы расположены в архивах вычислительной системы так, как задано в примерах 4.12 и 4.13.

Пример 4.18. Трансляция и счет программы примера 4.17 на БЭСМ-6.

```
*NAME Использование библиотеки
*CALL EXECUTE
ИМЯ: СЧЕТ;
begin external AA=EXTA, XX=AX, BB=B, (55200);
. . .
end
eop
*PERSO: <ИС-библиотека>
*READ DRUM
*PERSO:55300, CONT
*MAIN СЧЕТ
*EXECUTE
    <входные данные для программы СЧЕТ>
*END FILE
```

Пример 4.19. Трансляция и счет программы примера 4.17 на ЕС ЭВМ.

```
// EXEC SIMCLG
//SIMULA.SYSIN DD *
ИМЯ: MODEL;
begin external AA=EXTA, XX=AX, BB=B, (SIMLIB);
. . .
end
eop
/*
//SIMLIB DD DSN=CLASSLIB(INFLIB),DISP=SHR,
// UNIT=SYSDA,DCB=BLKSIZE=1024,VOL=SER=USERTOM
//LKED.SYSIN DD *
    INCLUDE LIB(NEWLIB)
/*
//LIB DD DSN=CLASSMOD,DISP=SHR,
// VOL=SER=USERTOM,UNIT=SYSDA
//GO.SYSIN DD *
    <входные данные для программы MODEL>
/*
```

З а м е ч а н и е. В примере 4.19 можно избежать употребления управляющего предложения редактора связей INCLUDE, если при записи раздела NEWLIB в библиотеку CLASSMOD указать для этого раздела посредством предложения ALIAS альтернативные имена, совпадающие с именами его входных точек. В этом случае раздел NEWLIB будет подключен редактором связей с помощью механизма автовызова [16].

## 4.4. Средства связи с программами на других языках

Трансляторы с языка симула-67 для БЭСМ-6 и ЕС ЭВМ позволяют организовать взаимодействие симула-программы с программами, полученными с помощью других систем программирования. Эти программы будем называть внешними. Единственное требование, предъявляемое к внешним программам,— это соблюдение принятых в мониторной системе Дубна [18] (на БЭСМ-6) и в ОС ЕС ЭВМ [27] стандартных соглашений о связях. Наиболее просто организуется взаимодействие с программами на фортране [25, 13], поскольку компиляторы с фортрана обеспечивают выполнение упомянутых соглашений. Во внешних программах, написанных на автокодах, например, на мадлене [18], бемше [20], на языке ассемблера [22], выполнение соглашений о связях должен обеспечить пользователь.

Вызов внешних программ выполняется с помощью процедур или процедур-функций, а информационное взаимодействие осуществляется через параметры или общие блоки памяти (*common-блоки*). В разделе 4.4.1 рассматриваются задание обращений к внешним программам и правила передачи параметров, а в 4.4.2 — описания общих блоков.

**4.4.1. Вызов внешних программ.** В процессе работы симула-программы можно обращаться к программам на других языках, выполняющим стандартные соглашения о связях. В симула-программе ссылки на них задаются с помощью описаний процедур. В заголовке процедуры указывается внутреннее имя процедуры, используемое в симула-программе, задается совокупность формальных параметров и их спецификаций. Телом процедуры является оператор вида

```
code (fortran, <имя внешней программы>);
```

Обращение к внешней программе выполняется с помощью оператора процедуры или указателя функции. Тип процедуры функции определяется указателем типа в ее описании; динамической проверки корректности результата, получаемого от внешней программы, не производится.

Фактическими параметрами внешних процедур могут быть: переменные, константы, выражения, массивы,— причем для переменных (простых и с индексами) в процедуру передаются их адреса.

Выражения, употребленные в фактических параметрах, вычисляются в момент обращения и адреса значений передаются внешней программе. Константы копируются, и адреса копий передаются внешней программе. Для параметров-текстов во внеш-

ную программу передается адрес первой литеры. При передаче массива в качестве параметра передается адрес первого элемента массива. При этом следует учитывать, что элементы многомерных массивов располагаются по столбцам, т. е. так же, как и в языке фортран.

Способ представления данных целого, вещественного, булевского типов и кодировка литер текста в симула-программах совпадают с принятыми в фортране-IV (на ЕС ЭВМ) и фортран-Дубна (на БЭСМ-6).

Так, например, целые значения в симула-программах на ЕС ЭВМ представляются как значения типа `integer*4` в фортране-IV, вещественные — как `real*4`, а простым переменным типа `boolean` соответствует тип `logical*4`.

**З а м е ч а н и е.** На ЕС ЭВМ в массивах типа `character` и `boolean` для хранения значений каждого элемента отводится 1 байт.

**П р и м е р 4.20.** Для того чтобы получить возможность обращаться по имени ПЕРЕМЕЩЕНИЕ к внешней программе MOVE, имеющей 3 вещественных параметра, необходимо задать описание процедуры в следующем виде:

```
procedure ПЕРЕМЕЩЕНИЕ (X, Y, N); real X, Y, N;  
code (fortran, MOVE);
```

а сам вызов может иметь вид

```
ПЕРЕМЕЩЕНИЕ (3.26, A [I], X+5);
```

В компиляторе симула-67 для БЭСМ-6 реализованы два способа вызова внешних процедур: статический и динамический. Программы, вызываемые статически, загружаются в оперативную память одновременно с той программой, в которой описаны эти внешние программы. Программы, вызываемые динамически, загружаются лишь в момент обращения к ним, и занимаемая ими память освобождается после возврата в вызывающую программу. Симула-транслятор для ЕС ЭВМ допускает только статический вызов внешних программ.

Динамический способ вызова внешних программ имеет модификацию: вызов с фиксацией. В этом случае освобождение памяти производится только по явному указанию программиста. Последний способ дает экономию машинного времени при многократном использовании программы, но возлагает на программиста ответственность за освобождение занимаемой ею памяти. Декларации классов вызывать динамически нельзя. Можно, однако, оформить тело декларации класса в виде внешней процедуры и вызывать ее динамически.

Указание о динамическом способе загрузки внешних программ дается в операторе `code` после имени программы: ключ-

чевое слово `load` задает динамическую загрузку внешней программы в момент обращения к ней и удаление ее из памяти при возврате в вызывающую симула-программу, а комбинация `load, fix` — динамическую загрузку с фиксацией. Например, для обеспечения динамической загрузки программы `MOVE` из примера 4.20 достаточно записать оператор `code` в виде

`code (fortran, MOVE, load);`

Динамически (в том числе с фиксацией) загружать можно и автономно оттранслированные процедуры на языке симула-67. Для таких процедур в операторе `code` вместо ключевого слова `fortran` нужно употреблять слово `simula`, если вызываемая процедура не обращается к глобальным переменным. Если динамически загружается процедура, использующая глобальные переменные, то в операторе `code` указывается только внешнее имя процедуры и способ загрузки. Например, описание

`procedure ЗАПИСЬ; code (WRITE, load, fix);`

задает для процедуры `ЗАПИСЬ` динамическую загрузку с фиксацией.

Оператор `delete` предназначен для снятия фиксации с внешних, динамически загружаемых программ, и удаления их из оперативной памяти.

Синтаксис: `delete (<список имен>)` .

Аргументами служат имена внешних процедур, в телах которых есть указание о динамической загрузке с фиксацией.

При использовании фиксации динамически загружаемых программ следует иметь в виду, что загрузка таких программ производится на свободное место поля данных в момент первого обращения к ним, поэтому фактическое удаление их из памяти производится только в том случае, если при выполнении программы между операторами вызова и возвратом из процедуры (оператором `delete`) не встречается порождающих выражений, в противном случае освобождение этой памяти выполняется при «сборке мусора».

4.4.2. Размещение данных в общих блоках. Входной язык компилятора содержит средства, позволяющие размещать данные для программы (простые переменные, массивы) в общих блоках. Общий блок представляет собой область памяти, выделяемую под данные в момент загрузки объектной программы по правилам, принятым в языке фортран. Данным, размещаемым в общих блоках, не присваиваются начальные значения при входе в блок, содержащий описания этих данных,

а при выходе из него данные сохраняют последние присвоенные значения. В этом смысле данные из общих блоков аналогичны собственным переменным языка алгол-60.

Использование общих блоков дает удобные средства для передачи информации внешним программам, особенно программам, написанным на фортране. Общие блоки, используемые в симула-программе, должны быть определены в фортран-программе и иметь соответствующие имена.

Указание о размещении данных в общем блоке задается описанием вида

**common /X/ A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>;**

где X — имя общего блока, а каждая из конструкций A<sub>i</sub> (i = 1 ÷ n) может быть либо идентификатором простой переменной, либо именем массива с указанием его длины.

Пример 4.21. Общий блок с именем X длиной 1300 слов может быть задан в симула-программе следующим описанием.

**common /X/ A (100), B (200), H (1000);**

В фортран-программе ссылаться на этот **common**-блок можно предложением

**COMMON /X/ A (100), B (200), H (1000);**

Длина многомерных массивов может задаваться так же, как в фортране, — указанием максимальных длин по каждому измерению, например:

**common /X/ A (100), B (10, 20), H (10, 10, 10);**

Описание общего блока должно быть помещено в том же блоке программы, где описаны упомянутые в нем идентификаторы, причем до описаний этих идентификаторов.

Пример 4.22.

**common /T/A, B, C (2, 2), E (4);**

**real A, B; array C, E [—1 : 0, 4 : 5];**

Эти описания задают размещение переменных A, B и двумерных массивов C и E в общем блоке с именем T.

В общих блоках можно размещать и массивы с переменными границами. В этом случае при загрузке программы память под массив отведется в соответствии с длиной массива, указанной в описании общего блока, а доступ к элементам массива будет производиться с учетом значений граничных пар в описании массива.

В программах, откомпилированных в отладочном режиме, при входе в блок производится проверка соответствия длины



массива, определяемой граничными парами, и длины, объявленной для него в описании общего блока. Если последняя окажется меньше, то выдается предупреждающее сообщение и выполнение программы продолжается. В отладочном режиме для всех массивов, в том числе и для массивов из общих блоков, производится проверка корректности значений индексов в индексных выражениях.

#### З а м е ч а н и я.

1. Идентификаторы, использованные в общем блоке, могут не иметь обычного описания. В этом случае, по умолчанию, им присваиваются типы данных и размерность массивов по правилам, принятым в языке фортран.

Например, описание

```
common /C/ I, L, K (10, 20), T (100); (1)
```

эквивалентно следующим описаниям (при условии, что I, L, K, T — не описаны в блоке, содержащем описание (1)):

```
common /C/ I, L, K (10, 20), T (100);  
integer I, L; integer array K [1 : 10, 1 : 20];  
real array T [1 : 100];
```

2. Массивы, размещаемые в общих блоках, не могут использоваться в качестве фактических параметров в процедурах `discrete`, `histo`, `histd`, `linear`.

3. В симула-программах на БЭСМ-6 можно обращаться к неименованной общей области, задав описание вида

```
common A1, A2, ..., AN;
```

В листинге загрузки такой общий блок имеет имя `**`.

4. Иногда на БЭСМ-6 возникает необходимость расположить общий блок с начала листа оперативной памяти. Такие блоки используются, например, при обменах с внешней памятью. В симула-программах такое расположение общего блока задается конструкцией (P), записываемой после имени `common`-блока. Например, описание

```
common /ОБМЕН/ (P) БУФЕР (1024);
```

задает общий блок с именем ОБМЕН, в котором размещается массив БУФЕР длиной 1024 слова. Память под этот общий блок будет отведена с начала листа.

## 4.5. Отладка симула-программ в пакетном режиме

В трансляторе симула-ЕС основными средствами пакетной отладки являются операторы, задающие трассировку программы и отладочный ввод-вывод; операторы, выполняющие печать

символьного дампа поля данных (ПД) и управляющего списка. В реализации на БЭСМ-6 имеется только оператор трассировки.

При применении операторов отладки компиляция программы должна производиться в отладочном режиме (см. 4.1.2, 4.1.3). В этом режиме в объектные программы (эквивалент исходной симула-программы на языке машины) вносятся средства индикации динамических ошибок в терминах входного языка и средства трассировки программ. Печать символьного дампа производится при возникновении ошибки во время выполнения симула-программы независимо от того, в каком режиме (счетном или отладочном) она выполнялась.

4.5.1. Трассировка программ. Средства трассировки позволяют следить за динамикой исполнения симула-программы. Трассировкой в дальнейшем будем называть отладочную печать, выдаваемую при прохождении управления через характерные точки программы — узлы трассировки.

В трансляторах на ЕС ЭВМ и БЭСМ-6 определены следующие виды узлов трассировки, которым присвоены номера.

1. Операторы перехода и условия в условных операторах.
2. Входы в блоки, выходы из блоков.
3. Обращения к процедурам, выходы из процедур.
4. Операторы detach, resume.
5. Активные фазы процессов.
6. Генерация объектов.
7. Завершение работы объектов.
8. Выдача управляющего списка.

Последний вид трассировки (номер 8) реализован только для транслятора на ЕС ЭВМ.

Для каждого вида имеется счетчик числа трассировок данного вида. Заполнение и изменение счетчиков производится операторами трассировки. При прохождении управления через узел трассировки положительное значение соответствующего счетчика уменьшается на единицу. Нулевое или отрицательное значение счетчика отменяет трассировку данного вида.

Задание режима трассировки выполняется с помощью оператора

trace (N, K);

где N — номер, соответствующий виду узла, K — новое значение счетчика данного вида.

В качестве параметров N и K могут быть использованы арифметические выражения, значения которых приводятся к целому типу. Если значение параметра  $K \leq 0$ , то выдача сообщений о прохождении узла данного вида не производится. Выполнение оператора трассировки по выбранному виду узла

в случае его повторного использования сводится к выполнению предписаний последнего оператора. Для отмены трассировки по выбранному узлу до исчерпания счетчика необходимо использовать оператор `trase` с параметром  $K \leq 0$ . Каждый вид трассировки можно использовать независимо друг от друга, кроме трассировки по узлу вида 8, которая должна использоваться только совместно с трассировкой по узлу вида 5.

Существует возможность задавать ( $K \geq 1$ ) или отменять ( $K \leq 0$ ) трассировку одновременно по всем 8 видам одним оператором `trase (N, K)`, в котором значения параметра  $N = 0$ .

Ни один из видов трассировки не распространяется на встроенные процедуры, процедуры-атрибуты системных классов и процедуры, написанные на других языках программирования.

Каждое трассировочное сообщение содержит номер оператора исходной симула-программы, что позволяет идентифицировать оператор, вызвавший это сообщение.

Рассмотрим сообщения, выдаваемые при прохождении каждого из видов узлов.

1. Операторы перехода и условия в условных операторах.

\* <N оператора> \* ПЕРЕХОД НА \* <N оператора> \*

или

\* <N оператора> \* УСЛОВИЕ  $\left\{ \begin{array}{l} \text{TRUE} \\ \text{FALSE} \end{array} \right\}$

2. Входы в блоки, выходы из блоков.

При входе в блок:

\* <N оператора> \* ВХОД В БЛОК <N блока>

При выходе из блока:

\* <N оператора> \* ВЫХОД ИЗ БЛОКА <N блока> ,

причем это сообщение выдается только при выходе из блока через завершающий его символ `end`.

3. Обращения к процедурам, выходы из процедур.

При обращении к процедуре

\* <N оператора> \* ВЫЗОВ <имя процедуры> ПАРАМЕТРЫ:

<параметры процедуры>.

Значения фактических параметров процедуры печатаются вместе с идентификаторами соответствующих формальных параметров. Значения параметров, вызываемых по наименованию, печатаются лишь в том случае, если соответствующий фактический параметр является простой переменной или константой. В противном случае на месте значения печатается слово

**ВЫРАЖЕНИЕ**, т. е. при трассировке, во избежание возможных побочных эффектов, не производятся вычисления значений фактических параметров выражений и переменных с индексами, вызываемых по наименованию.

При выходе из процедуры через завершающий ее символ **end**:

**\*<N оператора>\*ВОЗВРАТ ИЗ <имя процедуры>**

**С РЕЗУЛЬТАТОМ:** <результат>, причем часть сообщения **С РЕЗУЛЬТАТОМ:** <результат> печатается только для процедуры-функции и результат печатается в формате, соответствующем типу процедуры-функции. Если процедура-функция имеет ссылочный тип, то в качестве результата печатается либо **NONE**, либо информация об объекте, на который ссылается процедура-функция, в виде

**ОБЪЕКТ <имя класса> <N объекта> <атрибуты>.**

4. Операторы **detach**, **resume**.

При выполнении оператора **detach**:

**\*<N оператора>\*ОТКРЕПЛЕНИЕ:**  $\left\{ \begin{array}{l} \text{САМОСТОЯТЕЛЬНЫЙ} \\ \text{ПРИКРЕПЛЕННЫЙ} \end{array} \right\}$   
**ОБЪЕКТ <N объекта> <атрибуты>**

При выполнении оператора **resume**:

**\*<N оператора>\*ВОЗОБНОВЛЕНИЕ: САМОСТОЯТЕЛЬНЫЙ  
ОБЪЕКТ <N объекта>.**

При возврате главного управления на оператор, следующий за операторами **detach** или **resume**:

**\*<N оператора>\*УПРАВЛЕНИЕ ПОЛУЧИЛ ОБЪЕКТ  
<N объекта> <атрибуты>.**

5. Активные фазы процессов

При активных фазах процессов, являющихся подклассом системного класса **process**:

**\*<N оператора>\*TIME= <системное время> ПРОЦЕСС  
<имя класса> <N объекта> <атрибуты>.**

При исполнении планирующих операторов:

а) **hold**:

**\*<N оператора>\*HOLD НА <время задержки> ПРОЦЕСС  
<имя класса> <N объекта> <атрибуты>**

б) **passivate**:

**\*N оператора\*PASSIVATE ПРОЦЕСС  
<имя класса> <N объекта> <атрибуты>**

в) cancel:

\* <N оператора> \* CANCEL ПРОЦЕСС  
<имя класса> <N объекта> <атрибуты>

г) wait:

\* <N оператора> \* WAIT В ОБЪЕКТЕ  
<имя класса> <N объекта> <атрибуты>.

6. Генерация объектов.

При генерации объектов с помощью генератора объектов:

\* <N оператора> \* ГЕНЕРАЦИЯ: ОБЪЕКТ  
  
<имя класса> <N объекта> <атрибуты>.

При входе в квазипараллельную систему:

\* <N оператора> \* ГЕНЕРАЦИЯ: КПС <имя КПС> <атрибуты>.

7. Завершение работы объектов.

При прохождении управления через завершающий end декларации класса:

\* <N оператора> \* ЗАВЕРШЕНИЕ: ОБЪЕКТ  
<имя класса> <N объекта> <атрибуты>.

8. Выдача управляющего списка.

Этот вид трассировки может использоваться только одновременно с использованием трассировки по узлу типа 5 (активные фазы процессов). В этом случае сразу после трассировочного сообщения, соответствующего узлу трассировки вида 5, печатается информация о процессах, представленных в управляющем списке. Управляющий список печатается так же, как и при использовании оператора evq (0, 0) (см. 4.5.4).

Рассмотрим в качестве примера следующую программу.

Пример 4.23.

0.0 ИМЯ: ТЕСТ, РЕЖ=ОТЛ;

1.0 begin real ТМОДЕЛ;

1.1 ТМОДЕЛ:=400; trace (0, 2); trace (3, 4); simulation

2.0 begin ref (ПРИБОР) ССЫЛКА; real ТРАБОТЫ;

array МАТРИЦА [1:3, 1:3]; text ТЕКСТ1;

3.0 procedure dump; code (SYSTEM, SDUMP);

4.0 process class ПРИБОР;

5.0 begin integer СТР, СТОЛ;

5.1 for СТР:=1 step 1 until 3 do

5.1 for СТОЛ:=1 step 1 until 3 do

5.1 МАТРИЦА[СТР, СТОЛ]:=СУММА(СТР, СТОЛ) \*  
0.375;

```

5.2 hold (ТРАБОТЫ); dump; evq (0, 0); end;
6.0 integer procedure СУММА (I, J); integer I, J;
6.1 СУММА:=I + J;
2.1 ТЕКСТ1:—text ('ЗАПОЛНЕНИЕ МАТРИЦЫ');
2.2 ТРАБОТЫ:=184.592; ССЫЛКА:—new ПРИБОР;
2.4 activate ССЫЛКА; hold (ТМОДЕЛ);
2.7 end

```

```
1.5 end
```

```
0.3 eop
```

В этой программе оператором trace (0, 2) (оператор 1.2) задается трассировка по всем 8 видам узлов с числом прохождений узлов каждого вида равным 2. Следующим оператором trace (3, 4) изменяется значение счетчика третьего вида (вход и выход из процедуры), которое теперь становится равным 4.

При выполнении программы на ЕС ЭВМ будут напечатаны следующие трассировочные сообщения:

\*\*\*УПРАВЛЕНИЕ В ПРОГРАММЕ ТЕСТ\*\*\*

```

*1.4*ГЕНЕРАЦИЯ: КПС SIMULAT ССЫЛКА=NONE
ТРАБОТЫ=0.00000E+00 МАТРИЦА=NONE (МАССИВ)
ТЕКСТ1=NOTEXT SQS=NONE REFTП=NONE SUCS=
=NONE PREDs=NONE EV=NONE T1=FALSE
*2.3*ГЕНЕРАЦИЯ: ОБЪЕКТ ПРИБОР 4 СТР=0 СТОЛ=0
EVENT=NONE TERM1=FALSE SUCC=NONE PREDD=
=NONE
*5.1*ВЫЗОВ СУММА ПАРАМЕТРЫ: I=1 J=1
*5.1*ВОЗВРАТ ИЗ СУММА С РЕЗУЛЬТАТОМ: 2
*5.1*ВЫЗОВ СУММА ПАРАМЕТРЫ: I=1 J=2
*5.1*ВОЗВРАТ ИЗ СУММА С РЕЗУЛЬТАТОМ: 3
*5.2*HOLD НА 1.84591E+02 ПРОЦЕСС ПРИБОР 4 СТР=4
СТОЛ=4 EVENT=EVENTNO 0 TERM1=FALSE
SUCC=NONE PREDD=NONE
*2.5*TIME=0.00000E+00 ПРОЦЕСС SIMULAT ССЫЛКА=
=ПРИБОР 4 ТРАБОТЫ=1.84591E+02 МАТРИЦА=
=03E5A4 (МАССИВ) ТЕКСТ1=ЗАПОЛНЕНИЕ
МАТРИЦЫ SQS=HEAD 2 REFTП=SIMULAT SUCS=
=EVENTNO 3 PREDs=NONE EV=NONE T1=FALSE
**УПР. СПИСОК
0.00000E+00** (ТЕСТ) ПРОЦЕСС SIMULAT ССЫЛКА =
=ПРИБОР 4 ТРАБОТЫ=1.84591E+02 МАТРИЦА=
=03E5A4 (МАССИВ) ТЕКСТ1=ЗАПОЛНЕНИЕ МАТРИЦЫ
SQS=HEAD 2 REFTП=SIMULAT SUCS=EVENTNO 3
PREDs=NONE EV=NONE T1=FALSE
1.84591E+02*5.2*ПРОЦЕСС ПРИБОР 4 СТР=4 СТОЛ=4
EVENT=EVENTNO 0 TERM1=FALSE SUCC=NONE

```

PREDD=NONE

\*5.5\*ЗАВЕРШЕНИЕ: ОБЪЕКТ ПРИБОР 4 СТР=4 СТОЛ=4  
EVENT=EVENTNO 0 TERM1=FALSE SUCC=NONE  
PREDD=NONE

З а м е ч а н и я. Первые два пункта этих замечаний справедливы только для реализации на ЕС ЭВМ.

1. Перед первым трассировочным сообщением любого типа печатается

\*\*\*УПРАВЛЕНИЕ В ПРОГРАММЕ <имя программы>\*\*\* (1)

где <имя программы> — это имя симула-программы, в которой осуществляется трассировка. Если в процессе выполнения управление попадет в другую автономно откомпилированную программу, и в ней тоже будет трассировка (а это произойдет, если программа откомпилирована в отладочном режиме), то перед первым трассировочным сообщением для этой программы будет напечатано сообщение (1) с новым именем программы. При каждом переходе управления из одной программы в другую печатается такое сообщение.

2. В трассировочных сообщениях информация об атрибутах распечатывается так же, как и в символьном дампе (см. 4.5.3), кроме переменных ссылочного типа, для которых значение печатается в виде

<имя класса> <N объекта>

3. Каждый генерируемый объект, за исключением объектов, являющихся частью блока с префиксом, идентифицируется порядковым номером, который выводится на печать в отладочных сообщениях.

4.5.2. О т л а д о ч н ы й в в о д - в ы в о д. Традиционным средством отладки, широко применяемым программистами, является печать значений интересующих величин в различных точках программы. Такая печать может быть организована с помощью стандартных средств ввода-вывода, определенных в языке симула-67. Однако для отладочной печати, где формат выводимых данных не имеет существенного значения, более удобно использовать оператор print, позволяющий выводить значения нескольких величин в стандартном формате, определяемом типом выводимой величины. Этот оператор имеет следующий синтаксис:

print (<список вывода>);

где <список вывода>::= <элемент списка вывода> |

<список вывода>, <элемент списка вывода>

<элемент списка вывода>::= <выражение> |

```

<оператор процедуры>|<перевод строки>|
<идентификатор массива>
<перевод строки>::=/

```

При исполнении оператора `print` производится последовательный вывод на системный файл `sysout` значений для каждого элемента списка вывода.

Вывод выражения состоит в преобразовании его значения из внутреннего представления в последовательность символов и занесении этой последовательности в буфер вывода, представляющий строку печатающего устройства (см. 3.3.5). Фактический вывод буфера на печать производится при его заполнении. Форматы вывода, применяемые в операторе `print`, задаются базовыми операторами вывода, определенными в языке симула-67 [10, 26]. Например, оператор

```
print (I, R, B, C, T);
```

где `I, R, B, C, T` — выражения, имеющие тип `integer, real, boolean, character` и `text` соответственно, эквивалентен следующим операторам: `outint (I, 11); outreal (R, 8, 13); outchar (if B then "T" else "F"); outchar (C); outtext (T);`.

Таким образом, форматы вывода значений выражений в операторах отладочного вывода определяются их типами.

Кроме перечисленных типов выражений в операторе `print` может быть указано выражение типа `ref`. При выводе объектного выражения выполняются следующие действия: выводится на печать текущее содержимое буфера вывода, а затем, начиная с новой строки, печатается информация об объекте, на который ссылается данное объектное выражение. Например, в результате работы следующего фрагмента программы:

```

ref (A) RA;
class A (X); integer X;
  begin real Z; text T;
    Z:=X+10; T:—text ('ПРОВЕРКА');
  end;
RA:—new A (149); print ('ЗНАЧЕНИЕ RA:', RA);

```

будет получена такая выдача:

```

ЗНАЧЕНИЕ RA:
ОБЪЕКТ A 1 X=149 Z=159 T=ПРОВЕРКА

```

При выводе массива производится последовательная распечатка (по столбцам) значений его элементов в формате, соответствующем типу массива. Например, оператор

```
print ('МАССИВ A:', A),
```



где массив A описан как **real array A [1:10, 1:7]**, эквивалентен оператору цикла:

```
for J:=1 step 1 until 10 do
  for I:=1 step 1 until 7 do
    outreal (A [I, J], 8, 13);
```

Если в роли элемента списка вывода использован оператор процедуры, то это эквивалентно обычному исполнению этого оператора. Например, оператор

```
print ('K=', K, 'L=', L, newline, M, L)
```

будет эквивалентен операторам

```
print ('K=', K, 'L=', L);
newline; print (M);
```

если идентификатор **newline** описан как

```
procedure newline; <тело процедуры>
```

Операторы процедур в роли элементов списка вывода в операторах **print** удобно использовать в том случае, если стандартный формат не удовлетворяет пользователя. Например, если булевы значения нужно выводить в виде символов 0 (**false**) и 1 (**true**), то можно описать процедуру

```
procedure OUTBOOL (B); boolean B;
outchar (if B then "1" else "0");
```

и использовать ее в операторах **print**:

```
print ('БУЛЕВСКАЯ ПЕРЕМЕННАЯ B=', OUTBOOL (B));
```

Для принудительного перевода строки при работе оператора **print** можно использовать символ / (косая черта), причем допускается присоединять его к следующему за ним или предшествующему ему элементу списка вывода. Употребление нескольких символов / подряд вызывает пропуск соответствующего количества строк. Действие символа / эквивалентно оператору **outimage**. Например, оператор

```
print ('ПЕРЕВОД СТРОКИ', 'ПРОПУСК ДВУХ СТРОК',
//, /A);
```

эквивалентен операторам

```
print ('ПЕРЕВОД СТРОКИ'); outimage;
print ('ПРОПУСК ДВУХ СТРОК'); outimage; outimage;
outimage; print (A);
```

Для более краткой записи операций ввода с системного файла `sysin` во входной язык компилятора введен оператор `read`, синтаксис которого аналогичен синтаксису оператора `print`, с тем отличием, что в роли элементов списка ввода не допускаются выражения и переменные типы «ссылка на объект». Семантику оператора `read` поясним на следующем фрагменте программы:

```
real array A [1:4, 1:10]; array D [1:6]; integer K, L;  
boolean B; character C; real R; integer I; text T;  
T:=blanks (20); read (I, R, //T, B, C, D [3], A);
```

оператор `read` эквивалентен следующей последовательности операторов:

```
I:=inint; R:=inreal; inimage; inimage;  
T:=intext (T.length);  
B:=INBOOL; C:=inchar; D[3]:=inreal;  
for L:=1 step 1 until 4 do  
  for K:=1 step 1 until 10 do  
    A [K, L]:=inreal;
```

где процедура-функция `INBOOL` описана следующим образом:

```
boolean procedure INBOOL;  
  begin character C;  
    ВВОД: C:=inchar; if C=" " then goto ВВОД;  
    if C="T" or C="1" then INBOOL:=true  
    else INBOOL:=false;  
  end INBOOL;
```

Из описания процедуры `INBOOL` видно, что при вводе булевских значений пробелы игнорируются, литеры "Т" и "1" соответствуют логическому значению **true**, а все остальные литеры — значению **false**.

Как видно из приведенного примера, семантика оператора `read` в части ввода данных типа **integer**, **real**, **boolean** совпадает с семантикой операторов бесформатного ввода языка симула-1, если в них не употребляется идентификатор `newline`. Использование операторов процедур в роли элементов списка ввода имеет тот же смысл, что и в операторе `print`. Например, в симула-программе аналогом оператора `read (I, newline, R)` языка симула-1 будет оператор `read (I, inimage, R)`.

**4.5.3. Символьный дамп поля данных.** Символьный дамп представляет собой распечатку поля данных (см. 4.1.1) в терминах языка симула-67, а также динамической и статической цепочек, отражающих динамику вызовов блоков (процедур) и совокупность доступных (в соответствии с правилами

ми локализации) экземпляров блоков (объектов, процедур). В нем указываются значения всех переменных симула-программы в форме, соответствующей их типу, с указанием имен переменных. Таким образом, символьный дамп— это отображение динамического состояния симула-программы в момент его выдачи.

Начинается символьный дамп с сообщения

\*\*\*\*\*СИМВОЛЬНЫЙ ДАМП\*\*\*\*\*,

а кончается сообщением

\*\*\*\*\*КОНЕЦ ДАМПА\*\*\*\*\*.

В дампе могут встречаться следующие сообщения:

1. <адрес> БЛОК <N блока>  
<переменные>,

где <адрес> — адрес экземпляра данных соответствующего блока в поле данных программы; <N блока> — порядковый номер блока, указанный в редактированной распечатке исходной программы; <переменные> — информация о значениях переменных, описанных в этом блоке.

2. <адрес> ОБЪЕКТ <имя класса> <N объекта>  
<атрибуты>,

где <N объекта> — уникальный порядковый номер объекта; <атрибуты> — информация об атрибутах данного объекта.

3. <адрес> КПС <имя класса в префиксе КПС> <атрибуты>
4. <адрес> МАССИВ <имя массива> <граничные пары>  
<элементы массива>

5. <адрес> ПРОЦЕДУРА <имя процедуры> <переменные>

6. <адрес> ПРИСОЕД. БЛОК  
АДРЕС ПРИСОЕД. ОБЪЕКТА <адрес>

7. <адрес> СВОБОДНАЯ ОБЛАСТЬ

В конце дампа печатаются сообщения:

ТЕКУЩИЙ БЛОК: <адрес текущего блока>

ДИНАМИЧЕСКАЯ ЦЕПОЧКА:

<адреса ЭД (экземпляров данных), составляющих динамическую цепочку>

СТАТИЧЕСКАЯ ЦЕПОЧКА:

<адреса ЭД, составляющих статическую цепочку>

Все переменные и атрибуты в символьном дампе печатаются следующим образом:

<имя переменной> = <значение>.

Значение переменной печатается в формате, соответствующем типу переменной. Значения переменных целого типа печатаются в целом формате; вещественного типа — в формате с плавающей точкой; ссылочного типа — в шестнадцатеричном

формате (эти значения являются адресами объектов, на которые указывают ссылочные переменные). Если значение ссылочной переменной равно **none**, то вместо адреса печатается слово **NONE**. Для логических переменных в качестве значений печатаются **TRUE** или **FALSE**. Значения текстовых переменных печатаются на отдельных строках в виде текста, причем если текст не уместится на строке, то остаток его переносится (начиная с 21 позиции) на следующую строку. Значение **notext** представляется при выдаче дампа словом **NOTEXT**. Значения элементов массивов печатаются подряд друг за другом по столбцам и в формате, соответствующем типу массива. Все адреса в символьном дампе печатаются в шестнадцатеричном формате.

Печать цепочек связи блоков начинается с текущего блока. Динамическую цепочку составляют блоки, динамически охватывающие текущий блок. Считается, что блок А динамически охватывает блок В, если

1) блок В получил управление от А;

2) блок С динамически охватывает В, а А динамически охватывает С.

Можно считать, что динамическая цепочка отражает историю выполнения программы.

Статическую цепочку составляют блоки, текстуально охватывающие текущий блок, т. е. те блоки, описания которых непосредственно доступны из текущего блока.

Рассмотрим в качестве примера следующую программу.

**Пр и м е р 4.24.**

```
1.0 begin procedure A (X); integer X;  
3.0 begin B (1); end;  
    procedure B (X); integer X;  
5.0 begin real Y; Y:=X/0; end;  
6.0 begin A (1); end  
end
```

При выполнении оператора  $Y:=X/0$  произойдет авост (деление на 0) и будет распечатан символьный дамп. Текущим в этот момент является 5-й блок. Статическая цепочка программы состоит из блоков 5 и 1, а динамическая цепочка из блоков 5—3—6—1.

Анализ состояния программы, который может быть легко проведен по символьному дампу, как правило, оказывает большую помощь в локализации допущенных в программе семантических ошибок.

Символьный дамп выдается транслятором при обнаружении ошибки в процессе выполнения симула-программы, если при ее

запуске на счет в поле PARM оператора EXEC был задан позиционный параметр DUMP. Выполнение программы на этом заканчивается. Кроме того, программист может получить символьный дамп отлаживаемой программы, задав в ней внешнюю системную процедуру DUMP с помощью описания:

**procedure dump; code (SYSTEM, SDUMP);**

и обратившись к ней в нужном месте программы оператором dump.

Например, при обращении к процедуре dump (пример 4.23 оператор 5.3) будет распечатан символьный дамп в следующем виде:

```
*****СИМВОЛЬНЫЙ ДАМП*****
03E500 БЛОК 1
      ТМОДЕЛ=4.00000E+02
03E52C КПС SIMULAT
      ССЫЛКА=03E6A0 ТРАБОТЫ=1.84591E+02 МАТРИЦА==
      =03E54A
      ТЕКСТ1=ЗАПОЛНЕНИЕ МАТРИЦЫ
      SQS=03E600 REFTП=03E52C SUCS=03E64C PREDS==
      =NONE
      EV=NONE T1=FALSE
03E54A МАССИВ МАТРИЦА [1:3, 1:3]
      7.50000E-01 1.12500E+00 1.50000E+00 1.12500E+00
      1.50000E+00 1.87500E+00 1.50000E+00 1.87500E+00
      2.25000E+00
03E600 ОБЪЕКТ HEAD 2
      SUCC=03E6FC PREDD=03E64C
03E64C ОБЪЕКТ EVENTNO 3
      ETIME=4.00000E+02 PROC=03E52C SUCC=03E600
      PREDD=03E6FC
03E6A0 ОБЪЕКТ ПРИБОР 4
      СТР=4 СТОЛ=4 EVENT=03E6FC TERM=FALSE
      SUCC=NONE PREDD=NONE
03E6FC ОБЪЕКТ EVENTNO 0
      ETIME=1.84591E+02 PROC=03E6A0 SUCC=03E64C
      PREDD=03E600
      ТЕКУЩИЙ БЛОК: 03E6A0
      ДИНАМИЧЕСКАЯ ЦЕПОЧКА: 03E52C 03E500
      СТАТИЧЕСКАЯ ЦЕПОЧКА: 03E52C 03E500
*****КОНЕЦ ДАМПА*****
```

В случае возникновения аварийной ситуации (деление на 0, неправильная адресация и т. д.) перед символьным дампом печатается информация, указывающая место ошибки и позволяющая понять ее причину:

АДРЕС ЗАГРУЗКИ ОБЪЕКТНОЙ ПРОГРАММЫ <адрес>  
PSW <содержимое PSW>  
АВОСТ В ПРОГРАММЕ <имя программы>  
ОТНОСИТЕЛЬНЫЙ АДРЕС АВОСТА <адрес>  
АБСОЛЮТНЫЙ — <адрес>  
КОД ПРЕРЫВАНИЯ <код>  
<расшифровка кода прерывания>  
РЕГИСТРЫ <содержимое регистров>  
ПЛАВАЮЩИЕ РЕГИСТРЫ 0—6 <содержимое регистров>  
ТАБЛИЦА ОТОБРАЖЕНИЯ  
<служебная информация>  
ОБЛАСТЬ СОХРАНЕНИЯ ОБЪЕКТНОЙ ПРОГРАММЫ  
<адрес> <содержимое области сохранения>

Содержимое регистров, таблицы отображения и области сохранения объектной программы представляет мало интереса для прикладных программистов, но очень полезно системному программисту, если ошибка обусловлена программными или аппаратными сбоями.

4.5.4. Управляющий список. *Управляющий список* — это системный набор, который служит для представления оси системного времени в программах, использующих средства моделирования, определенные в классе *simulation*. Членами управляющего списка являются уведомления о событиях, расположенные в нем по возрастанию времен, на которые запланировано выполнение этих событий. Уведомление о событии ссылается на некоторый объект класса *process* или его подкласса и содержит вещественное число, трактуемое как системное время события. Таким образом, уведомление отображает событие, запланированное на некоторое системное время, содержанием которого будет исполнение очередной активной фазы этого объекта. Уведомление, находящееся в начале управляющего списка, относится к процессу, активному (работающему) в данный момент. Управляющий список также иногда называют очередью событий (*event queue*).

Управляющий список дает достаточно полное представление о динамике выполнения программы, поэтому печать его содержимого весьма полезна при отладке программ, поскольку изменения в управляющем списке отражают результаты работы планирующих и управляющих операторов (*activate*, *reactivate*, *hold*, *cancel* и т. д.), используемых для организации квазипараллельного исполнения процессов.

Для печати информации о событиях, запланированных во временном интервале  $[T_n, T_k]$ , используется оператор отладки *evq* ( $T_n, T_k$ ). Если  $T_n$  будет больше  $T_k$ , то управляющий список

печататься не будет. Если  $T_n$  будет равно нулю, то полагается  $T_n = \text{TIME}$ , если  $T_k = 0$ , то  $T_k$  полагается равным бесконечности. Таким образом, распечатать весь управляющий список можно оператором `evq (0, 0)`.

Управляющий список печатается в следующем виде:

```
* <N оператора> * УПРАВЛЯЮЩИЙ СПИСОК (Tn, Tk):
<время> * <N оператора> * (<имя программы>) ПРОЦЕСС
    <имя класса> <N объекта> <атрибуты>
    :
<время> * <N оператора> * (<имя программы>) ПРОЦЕСС
    <имя класса> <N объекта> <атрибуты>
```

где `<время>` — запланированное время выполнения данного события; `<N оператора>` — точка реактивации (локальное управление) данного процесса (для активного процесса точка реактивации не определена и не печатается); `<имя программы>` — имя симула-программы, в которой определена декларация класса данного процесса.

Если процессы, представленные в управляющем списке, декларированы в одной программе, то ее имя печатается только у первого (текущего) процесса. Если же в управляющем списке найдется процесс, декларация класса которого находится в другой (автономно откомпилированной) программе, то ее имя будет напечатано в сообщении, соответствующем данному процессу.

В качестве примера рассмотрим приведенную выше программу (пример 4.23). При выполнении оператора `evq (0, 0)` (оператор 5.4) будет распечатан управляющий список программы в следующем виде:

```
* 5.4 * * * УПР. СПИСОК (0.00000E+00, 0.00000E+00):
1.84591E+02 * * (ТЕСТ) ПРОЦЕСС ПРИБОР 4 СТР=4 СТОЛ=4
    EVENT=EVENTNO 0 TEMP1=FALSE SUCC=NONE
    PREDD=NONE
4.00000E+02 * 2.5 * ПРОЦЕСС SIMULAT ССЫЛКА=ПРИБОР 4
    ТРАБОТЫ=1.84591E+02 МАТРИЦА=03Е5А4 (МАССИВ)
    ТЕКСТ1=ЗАПОЛНЕНИЕ МАТРИЦЫ SQS=HEAD 2
    REFTП=SIMULAT
    SUCS=EVENTNO 0 PREDS=NONE EV=NONE T1=
    =FALSE
```

#### 4.6. Диалоговая отладка симула-программ

Диалоговая отладка симула-программ, в операционной системе ОС ЕС, проводится с помощью символьного отладчика. Он позволяет проводить интерактивную отладку объектных программ, полученных при помощи системы программирования

симула-67/ЕС [7]. Диалог ведется с использованием имен, определенных в отлаживаемой симула-программе.

4.6.1. Основные возможности диалогового отладчика. Отладчик предоставляет следующие возможности:

- запуск объектной программы на счет;
- задание останова в заказанных местах программы;
- вывод на терминал значений идентификаторов, доступных из точки останова, включая дистанционные идентификаторы и переменные с индексами;
- просмотр на экране дисплея информации, выводимой на печать в процессе счета симула-программы (файл sysout, см. 3.3),
- продолжение выполнения программы,
- задание с дисплея режимов трассировки программ (см. 4.5.3).

Программный комплекс, реализующий отладчик, занимает объем оперативной памяти, равный 16 Кбайт, и настроен на работу с дисплеями типа ЕС-7066, ЕС-7920.

4.6.2. Директивы диалоговой отладки. Все директивы отладчика вводятся в ответ на приглашение, появляющееся на экране дисплея после завершения обработки предыдущей директивы. Приглашением служит знак равенства (=). Каждая директива отладчика должна завершаться нажатием клавиши ВВ, что служит сигналом к началу обработки директивы.

Если при исполнении директивы обнаруживается ошибка, то выдается сообщение об ошибке, директива игнорируется и выдается приглашение на ввод следующей директивы. Сообщения отладчика приводятся в Приложении 7.

Текст директивы начинается с 8 позиции. Именно в ней оказывается курсор после выдачи приглашения. Длина директивы не должна превышать 72 символа, включая символ «конец текста», появляющийся при нажатии клавиши ВВ.

Директивы состоят из поля операции и поля операндов, которые разделяются хотя бы одним пробелом. Некоторые директивы не имеют операндов. Ниже рассматриваются синтаксис и семантика директив диалоговой отладки.

Директива ИМЯ. Синтаксис:

<директива ИМЯ>::=ИМЯ <идентификатор>| ИМЯ

Семантика. С помощью этой директивы отладчику задается имя программы, в которой будет производиться поиск контрольной точки, устанавливаемой по директиве КТ (см. ниже). Эта директива используется при отладке программных комплексов, состоящих из нескольких автономно скомпилированных программ. Чтобы однозначно идентифицировать контрольную точку, отладчику необходимо указать имя автономно откомпили-



рованной программы, в которой она устанавливается. Программу, указанную при последнем исполнении директивы ИМЯ, будем называть текущей для отладчика программой, а ее имя — текущим именем.

При попытке изменения текущего имени на имя программы, отсутствующей в оперативной памяти, выдается сообщение **ОТСУТСТВУЕТ ИМЯ** <имя программы>, и директива игнорируется. Если задана директива ИМЯ без операнда, т. е. после набора кода директивы нажата клавиша ВВ, то на экран дисплея будет выведено текущее имя.

**Директива КТ. Синтаксис:**

<директива КТ>::=КТ <номер блока>.<номер оператора>| КТ <относительный адрес>

<номер блока>::=<целое без знака>

<номер оператора>::=<целое без знака>

**Семантика.** Эта директива позволяет установить контрольную точку — место в программе, при достижении которого выполнение программы будет прервано и управление перейдет к отладчику, который выдаст сообщение о достижении контрольной точки и приглашение для ввода директив пользователя. Директива КТ заносит информацию о контрольной точке в таблицу контрольных точек.

Существует два способа задания контрольных точек: по номеру и по относительному адресу.

В первом случае контрольная точка задается номером блока и номером оператора, которые определяются по нумерации, содержащейся в листинге симула-программы, выдаваемом при ее трансляции. Заметим, что запись номера блока может состоять не более чем из 3 цифр, а номера оператора — не более чем из 4 цифр. Этот способ может быть использован только для симула-программ, которые оттранслированы в отладочном режиме.

Если оператор, указанный при задании контрольной точки, отсутствует в текущей программе, то будет выдано сообщение об ошибке: **КТ НЕ НАЙДЕНА**. При попытке установить контрольную точку с помощью номера блока и номера оператора в программе, оттранслированной в счетном режиме, выдается сообщение **СЧЕТНЫЙ РЕЖИМ**.

При втором способе задания контрольной точки, в качестве операнда директивы указывается относительный шестнадцатеричный адрес, который определяется программистом по таблице относительных адресов операторов, выдаваемой на печать при трансляции программы. Этот способ может использоваться для программ, оттранслированных как в отладочном, так и в счет-

ном режимах. Относительный адрес должен быть кратен 2, в противном случае выдается сообщение об ошибке: АДРЕС НЕ КРАТЕН 2.

Число заказанных точек останова, занесенных в таблицу контрольных точек, не должно превышать 10. При попытке установления большего числа точек выдается сообщение: ЛИШНЯЯ КТ, и контрольная точка не устанавливается. Если устанавливаемая контрольная точка уже присутствует в таблице, то действие директивы эквивалентно пустому оператору.

Отметим, что поиск контрольной точки осуществляется в программе, имя которой является текущим. Для установления контрольной точки в другой программе необходимо выполнить директиву ИМЯ, в качестве операнда которой указывается имя нужной программы.

Д и р е к т и в а О Т М. Синтаксис:

<директива ОТМ>::=ОТМ <номер блока>.<номер оператора>|  
ОТМ <относительный адрес> | ОТМ  
<номер блока>::=<целое без знака>  
<номер оператора>::=<целое без знака>

Семантика. С помощью этой директивы осуществляется отмена ранее установленной контрольной точки в программе с текущим именем. Если указанная в директиве контрольная точка отсутствует в таблице, то выдается сообщение КТ НЕ НАЙДЕНА. Использование директивы ОТМ без операнда приводит к одновременной отмене всех ранее установленных точек.

Д и р е к т и в а И Д И. Директива не имеет операндов. Она позволяет продолжить выполнение программы пользователя с той точки, где оно было приостановлено. Выполнение продолжается до тех пор, пока не встретится контрольная точка или завершающий символ end главной программы. При достижении контрольной точки выдается сообщение

КТ <номер блока>.<номер оператора><имя программы>  
или

КТ <относительный адрес><имя программы>

после чего отладчик готов к получению директив пользователя. Достигнутая контрольная точка исключается из таблицы.

Д и р е к т и в а Т А Б. Используется для вывода на экран дисплея таблицы контрольных точек. В результате выполнения этой директивы на экран будут отображены установленные контрольные точки.

Д и р е к т и в а Н О В. Эта директива служит для повторного выполнения отлаживаемого комплекса программ. Использовать эту директиву можно в любой момент времени после запуска

отладчика. В результате выполнения директивы на экране дисплея появляется сообщение

**СИМВОЛЬНЫЙ ОТЛАДЧИК СИМУЛА-ЕС. ВЕРСИЯ <номер>  
ИМЯ ОТЛАЖИВАЕМОЙ ПРОГРАММЫ <идентификатор>**

и выдается приглашение. После этого отлаживаемая программа готова к выполнению, а отладчик — к получению директив.

**Директива К О Н.** Эта директива используется для завершения отладки. После выполнения директивы на экране появится сообщение **КОНЕЦ СЕАНСА**, и на печатающее устройство будет выдан протокол сеанса совместно с информацией, которую выводила на печать отлаживаемая программа.

**Директива П П.** Осуществляет выдачу на экран дисплея протокола загрузки программ, формируемого загрузчиком ОС ЕС [16]. В этом протоколе указываются имена программ, входных точек и соответствующие им начальные адреса памяти. Заметим, что протокол загрузки программ всегда выводится в протокол сеанса по директивам Н О В и К О Н, если, конечно, не была использована директива П Р О (см. ниже).

**Директива П В.** С помощью этой директивы можно вывести на экран дисплея информацию, накопившуюся в системном файле sysout. При повторном использовании директивы П В на экран будет выдаваться только та информация, которая накопилась в файле после выполнения предыдущей директивы П В. Выдача производится в режиме роллинга.

Для того чтобы приостановить выдачу на экран дисплея информации из файла sysout, необходимо нажать клавишу В В. После нажатия клавиши загорается сигнал Б Л К и выдача на экран информации приостанавливается. Для гашения Б Л К надо нажать клавишу В С Т, после чего задача будет находиться в состоянии ожидания. Если есть необходимость прекратить дальнейший вывод информации, то необходимо на экране набрать любой символ, отличный от пробела, и нажать клавишу В В. Для продолжения вывода информации необходимо нажать только клавишу В В, не набирая никакого символа.

**Директива П Р О.** Устанавливает признак, отменяющий выдачу информации с файла sysout в протокол сеанса, которая производится отладчиком при исполнении директив Н О В и К О Н. Заметим, что действие этого признака не распространяется на директиву П В, т. е. информация, выданная на экран при исполнении этой директивы, будет присутствовать в протоколе диалога. Директива П Р О действует вплоть до выдачи директивы Н О В.

**Директивы Т R A C E, E V Q, П Д.** Действия этих директив совпадают по форматам и действиям с операторами от-

ладки trace, evq, dump (см. 4.5). Для выдачи на экран результатов работы этих директив необходимо выполнить директиву ПВ.

**Директива В Ы В. Синтаксис:**

```
<директива В Ы В>::=В Ы В <список переменных>
<список переменных>::= <переменная> |
    <переменная>, <список переменных>
<переменная>::= <идентификатор> |
    <идентификатор>[<список индексов>] |
    <идентификатор>.<переменная> |
    <идентификатор>[<список индексов>].<переменная>
<список индексов>::= <индекс> | <индекс>, <список индексов>
<индекс>::= <идентификатор> |
    <идентификатор>.<идентификатор> | <целое>
```

**Семантика.** Эта директива предназначена для вывода на экран дисплея значений переменных, указанных в качестве операндов директивы и доступных в данной точке программы (точке останова).

Доступность переменной в данной точке программы определяется по правилам языка симула-67: переменная доступна, если она описана в одном из блоков, текстуально охватывающих данное место программы, или является атрибутом объекта, на который указывает объектное выражение в дистанционном идентификаторе. Если переменная недоступна, то выдается сообщение вида ИМЯ <идентификатор> НЕ НАЙДЕНО.

Необходимо отметить, что обращение к переменной интерпретируется слева направо. Например, при обращении вида

<идентификатор 1>.<идентификатор 2>

доступность <идентификатора 2> определяется его доступностью в контексте того класса, на объект которого указывает <идентификатор 1>. Если значение <идентификатора 1> равно none, то выдается сообщение об ошибке вида ЗНАЧЕНИЕ ОБЪЕКТНОГО ВЫРАЖЕНИЯ=NONE и обработка директивы заканчивается. Если на том месте в обращении, где может стоять только переменная ссылочного типа, стоит переменная другого типа, то это считается синтаксической ошибкой и выдается соответствующее сообщение (см. Приложение 7).

Так же как во входном языке компилятора симула-67/ЕС, идентификаторы различаются по первым 7 символам — остальные (если они есть) игнорируются. В записи идентификатора могут использоваться русские и латинские буквы и цифры.

Количество индексов не должно превышать восьми. Индексом может служить целая или вещественная переменная (в по-

следнем случае она приводится к целому виду), целая константа или дистанционный идентификатор, например: A[5], A[-3], A[I], A[X.I].

Заметим, что запись обращения к переменным ничем не отличается от принятой в языке симула-67. Единственное отличие — в том, что индекс массива не может сам быть индексируемой переменной, т. е. запись вида A[B[C, D]] является для отладчика недопустимой и, следовательно, будет распознава как синтаксически неверная с выдачей сообщения об ошибке.

Значения переменных выводятся на экран по одному в строке. Для переменных типа text выводится столько строк, сколько необходимо для выдачи всего текста, и на отдельной строке выводится сообщение вида ДЛИНА <целое>.

Если значение переменной типа text равно notext, то выводится NOTEXT.

Для переменных ссылочного типа выводится сообщение ОБЪЕКТ КЛАССА <имя класса><номер><16-ричный адрес>, или 000000,— если значение переменной равно none.

Значение переменной типа boolean выводится в виде TRUE или FALSE.

Если в записи директивы обнаружена синтаксическая ошибка, то выводится сообщение об ошибке вида ИСПОЛЬЗОВАНИЕ СИМВОЛА ' НЕДОПУСТИМО, где в кавычках стоит недопустимый в директиве символ. На этом обработка директивы заканчивается. Следует учитывать, что нераспечатываемый символ конца текста заменяется при обработке директивы на точку с запятой (;).

Директива УСТ. Синтаксис:

```
<директива УСТ>::=УСТ <список операторов присваивания>
<список операторов присваивания>::=
  <оператор присваивания>|
  <оператор присваивания><список операторов присваивания>
<оператор присваивания>::=<арифметический>|<ссылочный>|
  <присваивание текстового значения>|
  <присваивание текстовой ссылки>|
  <булевский>|<знаковый>
<арифметический>::=<переменная>:=<переменная>; |
  <переменная>:=<константа>;
<константа>::=<целое>|
  <вещественное с фиксированной запятой>|
  <вещественное с плавающей запятой>
<ссылочный>::=<переменная>:—<переменная>;
<присваивание текстового значения>::=
  <переменная>:=<переменная>; |
  <переменная>:=<текстовая константа>;
```

$\langle \text{текстовая константа} \rangle ::= \text{"последовательность литер"}$   
 $\langle \text{присваивание текстовой ссылки} \rangle ::= \langle \text{переменная} \rangle : -$   
 $\langle \text{переменная} \rangle ;$   
 $\langle \text{булевский} \rangle ::= \langle \text{переменная} \rangle : = \langle \text{переменная} \rangle ; |$   
 $\langle \text{переменная} \rangle : = \text{"T"} ; | \langle \text{переменная} \rangle : = \text{"F"} ;$   
 $\langle \text{знаковый} \rangle ::= \langle \text{переменная} \rangle : = \langle \text{переменная} \rangle , |$   
 $\langle \text{переменная} \rangle : = \text{"\langle алфавитно-цифровой знак \rangle"}$ .

Семантика. Эта директива выполняет простые операторы присваивания в смысле языка симула-67. Единственное синтаксическое отличие — в записи оператора присваивания для булевских переменных: значения **ИСТИНА** и **ЛОЖЬ** записываются как **"T"** и **"F"** соответственно, вместо **true** и **false**.

В последнем (или единственном) операторе точка с запятой может быть опущена. В этом случае следует нажать клавишу **ВВ** сразу после последнего знака оператора. Допускается произвольное количество пробелов перед первым и после последнего оператора.

Семантически действие директивы состоит в выполнении операторов. Единственное отличие от стандартов языка — в том, что если при выполнении присваивания текстовой ссылки значение переменной в левой части равно **notext**, то это считается ошибкой и выдается соответствующее сообщение об ошибке.

Следует отметить особенность интерпретации операторов: если для индексируемой переменной в правой части оператора было выдано сообщение **НЕ СОВПАДАЕТ РАЗМЕРНОСТЬ МАССИВА**  $\langle \text{идентификатор} \rangle$ , — или **ИНДЕКС ВЫХОДИТ ЗА ГРАНИЦЫ МАССИВА**  $\langle \text{идентификатор} \rangle$ , то переменной в левой части будет присвоено начальное значение, принятое в языке симула-67.

Типы переменных, стоящих в правой и левой частях оператора, должны быть приводимыми — в противном случае будет выдано сообщение об ошибке.

**4.6.3. Запуск отладчика.** Для использования отладчика необходимо, чтобы отлаживаемая программа, возможно, вместе с автономно оттранслированными программами, находилась в личной библиотеке пользователя или во временном наборе данных. В управляющих картах шага задания, задающих запуск отладчика, должно присутствовать **DD-предложение** с именем **OTLLIB**, указывающее раздел, в котором записан загрузочный модуль объектной программы. Если отлаживаемая программа состоит из нескольких автономно оттранслированных частей, то в **DD-предложении OTLLIB** нужно задать (с помощью сцепления наборов данных) все разделы, в которых записаны эти части.

Загрузка в оперативную память ЭВМ отлаживаемой программы, со всеми необходимыми программами интерпретирующей системы компилятора, осуществляется загрузчиком ОС ЕС.

Рассмотрим на примере запуск отладчика. Пусть требуется произвести отладку программы находящейся в наборе данных BIBL в разделе с именем ТЕСТ. Библиотека расположена на диске пользователя с именем SYSLIB. Для простоты предположим, что на этом же диске находится библиотека интерпретирующей системы компилятора симула-67/ЕС — SIM.IS. Тогда пакет запуска будет иметь вид

```
//EXEC PGM=SIMOTL,PARM=MAP,CALL,
  SIZE=n/⟨параметры для отлаживаемой программы⟩'
//STEPLIB      DD DSN=SIM.IS, DISP=SHR,
//  UNIT=SYSDA, VOL=SER=SYSLIB
//OTLLIB       DD DSN=BIBL(ТЕСТ), DISP=SHR,
//  UNIT=SYSDA, VOL=SER=SYSLIB
//SYSLIB       DD DSN=SIM.IS, DISP=SHR,
//  UNIT=SYSDA, VOL=SER=SYSLIB
//SYSLOUT      DD DSN=&LIST,
//  DISP=(NEW, PASS), UNIT=SYSDA, SPACE=(TRK, (25))
//SYSOUT       DD DSN=&OUT, DISP=(NEW, PASS),
//  UNIT=SYSDA, SPACE=(TRK, (25))
//SYSUTB       DD SYSOUT=A
//DD1          DD UNIT=⟨адрес дисплея⟩
//SYSIN        DD *
  ⟨данные, вводимые отлаживаемой программой⟩
/*
```

где *n* — размер (в байтах) динамической области основной памяти, которая может быть использована загрузчиком.

З а м е ч а н и е. Отлаживаемая программа может быть записана в библиотеку с помощью следующего шага.

```
// EXEC SIMCL
//SIMULA.SYSIN DD *
ИМЯ: ТЕСТ;
  ⟨текст симула-программы⟩
eor
//LKED.SYSLMOD DD DSN=BIBL(ТЕСТ), DISP=SHR,
//  UNIT=SYSDA, VOL=SER=SYSLIB
```

После ввода пакета запуска отладчика в машину и старта задачи на экране дисплея появится сообщение:

СИМВОЛЬНЫЙ ОТЛАДЧИК СИМУЛА-ЕС. ВЕРСИЯ ⟨номер⟩  
ИМЯ ОТЛАЖИВАЕМОЙ ПРОГРАММЫ ⟨имя⟩,

где в качестве <имени> будет указано имя, которое было задано при трансляции симула-программы в карте ИМЯ. После выдачи этих сообщений отладчик готов к работе. Можно вводить директивы.

Поясним назначение некоторых наборов данных, используемых при работе отладчика.

1. Наборы данных &LIST, &OUT с DD-именами SYSLOUT, SYSOUT.

Эти временные наборы используются для хранения протокола загрузки программ и системного файла вывода sysout. Их удобно размещать на системных дисках. Если размеры памяти, отведенные под эти наборы, будут недостаточны, то следует увеличить их с помощью задания большего значения параметра SPACE.

2. Выходной набор данных с именем SYSUTB.

В этот набор данных производится выдача протокола диалога. Для блокировки его выдачи на печатающее устройство, необходимо поставить карту //SYSUTB DD DUMMY.

В Приложении 8 приводятся пример симула-программы, использующей некоторые средства отладки, и протокол работы с этой программой с помощью диалогового отладчика.



## ГЛАВА 5

# ПРИМЕНЕНИЕ ЯЗЫКА СИМУЛА-67 ДЛЯ РАЗРАБОТКИ ПАКЕТОВ ПРИКЛАДНЫХ ПРОГРАММ

В данной главе рассматриваются некоторые вопросы применения языка симула-67 для построения пакетов прикладных программ (ППП), ориентированных на решение задач в определенной предметной области. Основу таких пакетов составляют библиотеки классов и процедур, отображающих существенные свойства основных типов объектов, характерных для данной предметной области. Эти библиотеки будем называть пакетами типовых моделей. Аппарат иерархий деклараций классов позволяет постепенно детализировать типовые модели в процессе их эксплуатации, отображая поэтапную конкретизацию представлений о сложных объектах и системах при их исследовании или проектировании. Средства раздельной компиляции (см. 4.3) дают возможность на каждом этапе фиксировать типовые модели в виде библиотек программ в архивах вычислительной системы и эффективно использовать их как основу для работы на следующем этапе.

В разделе 5.1 рассматривается методика разработки пакетов типовых моделей, основанная на применении средств иерархического описания классов (см. 1.3) и средств раздельной компиляции (4.3). Остальные разделы данной главы содержат описания и примеры применения нескольких конкретных пакетов, разработанных с помощью языка симула-67. Сведения о других пакетах и специализированных языках, созданных на базе языка симула-67, можно найти в работах [23, 37—40, 44—46].

### 5.1. Использование иерархий деклараций классов при разработке ППП

Основу пакета типовых моделей для некоторой предметной области составляют декларации классов и процедур, отображающие основные объекты, понятия, методы, выработанные

в данной области. В декларациях классов с помощью описаний атрибутов отображаются информационные признаки объектов, а их правила функционирования и другие алгоритмические свойства задаются в операторных частях деклараций и описаниях процедур, которые также могут быть атрибутами объекта.

Для наглядного, структуризованного отображения иерархических описаний с постепенной детализацией представления сложных объектов, а также систем понятий, имеющих древовидную структуру, удобно использовать аппарат префиксов, позволяющий организовать иерархии деклараций классов (см. 1.3). В этом случае структурность описания достигается тем, то если декларация некоторого класса В имеет префиксом имя некоторого другого класса А, то объекты класса В будут обладать как свойствами, задаваемыми декларацией класса А (типовыми свойствами), так и специфическими свойствами, задаваемыми в декларации класса В. В этом случае говорят, что класс В является подклассом класса А, а класс А — надклассом для В. Один класс может иметь несколько подклассов, т. е. может употребляться в качестве префикса для нескольких деклараций классов, а подкласс может, в свою очередь, выступать в качестве надкласса для нескольких других классов. Таким образом, несколько деклараций классов могут образовать дерево, корнем которого является декларация класса, не имеющего префикса и задающая свойства, общие для всех классов, образующих данную иерархию.

Привнесение свойств (структуры атрибутов и правил действий), задаваемых декларацией надкласса, в объекты подкласса выполняется путем формирования из деклараций надкласса и подкласса эквивалентной декларации, которая уже не имеет префикса и служит шаблоном для генерации объектов, принадлежащих к подклассу. В языке симула-67 этот процесс называется сочленением и выполняется по определенным правилам, составляющим алгоритм сочленения (см. 1.3). Считается, что сочленение выполняется до начала счета программы.

При формировании эквивалентной декларации для класса, имеющего префикс, происходит объединение атрибутов (списков формальных параметров, совокупностей спецификаций и описаний самого внешнего блока тела этого класса и его надкласса) и правил действий сочленяемых деклараций.

Операторы тела класса в эквивалентной декларации формируются путем помещения в тело надкласса на место символа **inner** операторов, составляющих тело подкласса. Если **inner** опущен, то считается, что он непосредственно предшествует символу **end**, замыкающему декларацию класса. Таким образом, в операторной части декларации надкласса удобно задавать

типовые действия, общие для целой совокупности классов объектов: операторы, предшествующие символу **inner**, будут использоваться до операторов, заданных в подклассах и соответствующих специфическим действиям, характерным для объектов каждого подкласса, а операторы, следующие за **inner**, — после них.

При создании пакетов типовых моделей в виде библиотек классов и процедур часто возникает необходимость обращаться к некоторому множеству глобальных величин, а также задавать некоторые стандартные действия, предвещающие и завершающие работу программ, использующих эти библиотеки. Эти стандартные действия могут зависеть от значений глобальных величин. В этом случае удобно сделать описания этих величин и деклараций классов и процедур атрибутами нового класса, идентифицирующего пакет, и задать в качестве его операторной части предвещающие и завершающие действия, разделив их символом **inner**.

Рассмотрим в качестве примера возможную структуру библиотеки классов и процедур для моделирования различных вариантов организации работы аэропорта. Декларации классов в такой библиотеке могут описывать типовые свойства объектов, существенные для работы аэропорта, а описания процедур могут задавать часто встречающиеся действия. Эти классы и процедуры могут использовать ряд общих для всей модели глобальных величин, таких, например, как максимальное и текущее количество пассажиров (МКП, ТКП), максимальное и текущее число самолетов (МКС, ТКС) в аэропорту, количество стоянок (КСТ), количество свободных стоянок (КСВСТ) и т. д.

Значения некоторых из этих величин могут задаваться как исходные данные для моделирования (например, МКП, МКС, КСТ), а значения других (ТКП, ТКС, КСВСТ) формируются в процессе работы конкретной модели аэропорта.

Декларации классов и описания процедур, задающие существенные в данном исследовании свойства самолетов, пассажиров и других объектов аэропорта вместе с описаниями, необходимые для работы глобальных величин, можно сделать атрибутами класса АЭРОПОРТ и выполнить их компиляцию с помощью следующей программы:

Пример 5.1.

ИМЯ: ПОРТ;

**begin**

**simulation class** АЭРОПОРТ (МКП, МКС, КСТ); **integer** МКП, МКС, КСТ;

**begin integer** ТКП, ТКС, КСВСТ, I; **ref** (САМОЛЕТ) **array** C [1 : МКС];

```

process class САМОЛЕТ (КОЛПАСС); integer КОЛПАСС;
  begin <описание типовых свойств самолета> end;
process class ПАССАЖИР;
  begin <описание правил действий пассажира> end;
procedure ВЫЛЕТ (X); ref (САМОЛЕТ) X; ...;
procedure ПОСАДКА; ...;
. . .
<описания других классов, процедур, переменных, массивов>
entry (САМОЛЕТ, ПАССАЖИР, ВЫЛЕТ, ПОСАДКА, ...
(УКНА));
ТКП:=МКП; ТКС:=МКС; КСВСТ:=0;
for I:=1 step 1 until МКС do C [I]:—new САМОЛЕТ (150);
<другие операторы, описывающие необходимые начальные
действия по организации работы модели АЭРОПОРТА>
inner;
<операторы, завершающие работу модели>
END АЭРОПОРТ;
entry (АЭРОПОРТ, (УКНА));
end

```

Через УКНА в данном примере обозначен указатель набора данных, в который компилятор запишет информацию о классах АЭРОПОРТ, САМОЛЕТ, ПАССАЖИР и их атрибутах.

При использовании созданного пакета типовых моделей основных объектов аэропорта, необходимо описать лишь дополнительные действия и объекты, отображающие специфику исследуемого варианта модели. Например, модель аэропорта с максимальным количеством пассажиров МКП=10000, максимальным числом самолетов МКС=250 и числом стоянок КСТ=75 может быть описана следующей программой.

Пример 5.2.

```

ИМЯ: MODEL 1;
begin external (УКНА);
  АЭРОПОРТ (10000, 250, 75)
  begin <дополнительные описания>;
  ВЫЛЕТ (C [1]); ВЫЛЕТ (C [20]);
  <операторы, описывающие конкретный вариант работы
аэропорта>
end
end

```

В соответствии с правилами сочленения при работе блока с префиксом АЭРОПОРТ (...) **begin ... end**; имеющегося в данной программе, будет сначала выполнено отведение памяти под атрибуты класса АЭРОПОРТ и величины, определенные

в дополнительных описаниях (память под массив  $S[1:MKC]$  будет выделена в соответствии с заданным значением  $MIC=250$ ), затем выполняются заданные в программе примера 5.1 операторы, описывающие начальные действия, после этого управление перейдет к операторной части блока с префиксом, а вслед за ней будут выполнены операторы, завершающие работу модели, которые заданы в декларации класса АЭРОПОРТ (пример 5.1).

Аппарат иерархий деклараций классов, составляющий основу языка симула-67, позволяет конкретизировать и детализировать описания типовых свойств объектов, заданных в библиотечных декларациях. Например, если при моделировании аэропорта с использованием средств, определенных в декларации класса АЭРОПОРТ, возникает необходимость описать самолеты типа ТУ-154, обладающие наряду с типовыми свойствами любых самолетов, заданными в декларации класса САМОЛЕТ (пример 5.1), рядом специфических свойств, то в блоке АЭРОПОРТ (...) **begin ... end** нужно поместить декларацию класса вида

**САМОЛЕТ class TU154,**

**begin** *<описание специфики самолетов ТУ154>* **end;**

где указываются дополнительные характеристики самолетов ТУ-154.

На основе класса АЭРОПОРТ можно построить целый ряд пакетов моделей, ориентированных на моделирование различных типов аэропортов. Например, специфика международных аэропортов может быть отражена в следующей декларации класса.

Пример 5.3.

**ИМЯ: INTER;**

**begin external (УКНА);**

**АЭРОПОРТ class МЕЖДУНАРОДНЫЙ (НАЗВАНИЕ);**

**text НАЗВАНИЕ;**

**begin**

**САМОЛЕТ class TU154; ...;**

**САМОЛЕТ class DC10; ...;**

**ПАССАЖИР class ТУРИСТ; ...;**

**. . .**

**process class ТАМОЖНЯ; ...;**

**entry (ТУ154, DC10, ТУРИСТ, ТАМОЖНЯ, (УКНВ));**

**Н<sub>м</sub>; inner; К<sub>м</sub>;**

**end;**

**entry (МЕЖДУНАРОДНЫЙ, (УКНВ));**

**end**

где через  $H_m$  и  $K_n$  обозначены начальные и конечные действия, выполняемые при моделировании любого международного аэропорта, а через УКНВ — указатель набора данных для записи информации о классе МЕЖДУНАРОДНЫЙ.

Как видно из примера 5.3, при трансляции программ, составляющих пакет типовых моделей для международных аэропортов, существенно используется информация о классах и процедурах, определенных в пакете АЭРОПОРТ (пример 5.1). В связи с этим, при внесении изменений в пакет АЭРОПОРТ, которые влияют на содержание информации, записываемой при его трансляции в набор данных, задаваемый указателем УКНА, необходимо перекомпилировать и все пакеты, построенные на его основе, в частности пакет МЕЖДУНАРОДНЫЙ (пример 5.3). К такого рода изменениям относятся любые коррективы, влияющие на атрибуты (параметры) классов и процедур, упомянутых в операторах entry. Исправления, вносимые в операторы, расположенные на уровне самых внешних блоков тел деклараций классов, могут не оказать влияния на выводимую информацию, в чем можно убедиться, сравнив соответствующие распечатки, выдаваемые компилятором. Заведомо не изменяют упомянутой информации изменения, вносимые в тела процедур и во внутренние блоки деклараций классов, а также добавления новых блоков в любое место программы пакета. Таким образом, если необходимо изменить операторную часть некоторого пакета без перетрансляции пакетов, построенных на его основе, изменения нужно вносить в виде блоков.

Приведем пример использования средств, введенных в примере 5.3, для моделирования работы аэропорта при максимальном числе пассажиров в интервале от 1000 до 8000 с шагом 1000, максимальном количестве самолетов — 150, числе стоянок — 100. Соответствующая программа будет иметь вид

Пример 5.4.

```
ИМЯ: INT1;  
begin external (УКНА); external (УКНВ); integer КПАСС;  
  for КПАСС:=1000 step 1000 until 8000 do  
    МЕЖДУНАРОДНЫЙ (КПАСС, 150, 100, 'ШЕРЕМЕТЬЕВО-2')  
  begin S;  
  end  
end
```

где через S обозначены операторы, описывающие конкретный вариант работы международного аэропорта. Исполнение операторов S в данной программе, будет обрамлено работой операторов  $H_m$  и  $K_m$ , так как в соответствии с правилами сочленения операторы S подставляются на место inner в примере 5.3.

Подобным образом при помощи иерархий деклараций классов и средств раздельной компиляции, реализованных в компиляторах с языка симула-67 для ЭВМ БЭСМ-6 и ЕС ЭВМ, могут быть разработаны многоуровневые системы пакетов (библиотек) моделей для широкого спектра предметных областей.

## 5.2. Пакет для моделирования непрерывно-дискретных систем

Средства моделирования, определенные в системном классе *simulation* языка симула-67 [10], ориентированы на описание имитационных моделей дискретных систем. В практике моделирования встречаются системы, в которых функционирование некоторых объектов (законы изменения их параметров  $Y_1, Y_2, \dots, Y_n$ ) удобно описывать с помощью систем дифференциальных уравнений вида

$$\dot{Y}_i = \Phi_i(Y_1, Y_2, \dots, Y_n, T), \text{ где } i = 1, 2, \dots, n, \quad (1)$$

Такие системы часто называют непрерывно-дискретными [8]. Некоторые языки моделирования, например недис [8], *gasp-IV* [43], содержат средства, позволяющие в удобной форме задавать системы уравнений вида (1). Программные комплексы, реализующие такие языки, обеспечивают решение системы дифференциальных уравнений, описывающих непрерывную часть модели, совместно с имитацией дискретной части.

Рассмотрим один из возможных способов построения средствами языка симула-67 на базе системного класса *simulation* пакета программ для моделирования непрерывно-дискретных систем. Другой способ решения этой задачи представлен в работе [37].

Предлагаемый в данной работе пакет программ реализован в виде класса *DISCONT*, атрибутами которого служат процедуры и классы, позволяющие представлять системы вида (1) и получать в процессе моделирования их численные решения.

Во время работы программы моделирования система уравнений (1) представляется с помощью списка объектов, каждый из которых содержит в качестве атрибутов ссылку на программу вычисления правой части  $\Phi_i(Y_1, Y_2, \dots, Y_n, T)$  и текущее значение  $Y_i$ , равное интегралу по системному времени от  $\Phi_i(Y_1, Y_2, \dots, Y_n, T)$ . Такое представление описано в [19] и используется в языке недис. Оно позволяет легко описывать системы с переменной структурой, поскольку состав упомянутого списка может изменяться в процессе моделирования. Элементы списка, которые в дальнейшем будем называть инте-

граторами, описываются в классе DISCONT с помощью следующей декларации класса:

```
link class INTGRL (Y, X); name X; real X, Y;  
  begin real Y1; comment Y1 — рабочая переменная;  
    this INTGRL.into(INTLIST); activate INTMET delay 0; (2)  
  end INTGRL;
```

Переменная  $Y$  в (2) используется для хранения текущего значения интеграла от выражения  $X$ , представляющего правую часть одного из уравнений системы (1). Конкретный вид правой части и начальные значения  $Y$  задаются при порождении интегратора. Предполагается, что ссылочная переменная INTLIST обозначает список интеграторов, а переменная INTMET указывает на процесс, который активизируется каждые  $DT$  единиц системного времени ( $DT$  — шаг интегрирования) и обновляет текущие значения интегралов путем просмотра списка INTLIST. Правила действия этого процесса определяются выбранным методом интегрирования. Действия, общие для всех методов с фиксированным шагом, заданы в классе DISCONT следующим образом:

```
process class INTEGRATION;  
  begin ref (INTGRL) I;  
    РАБОТА: hold (DT); if INTLIST.empty then  
      begin  
        ЖДУ: passivate; goto РАБОТА; (3)  
      end;  
      inner; goto РАБОТА;  
    end INTEGRATION;
```

Интегрирование методом Эйлера может быть теперь описано так:

```
INTEGRATION class EULIER;  
  begin  
    for I:—INTLIST.first, I.suc while I /= none do  
      inspect I do Y1:=Y + X * DT; hold (0)  
      for I:—INTLIST.first, I.suc while I /= none do I.Y:=I.Y1;  
    end EULIER;
```

Аналогичным образом могут быть заданы и другие методы интегрирования.

При моделировании средствами класса DISCONT объектов, описываемых системами вида (1) с начальными условиями  $Y_i(0) = Y0_i$   $i = 1, 2, \dots, n$ , нужно задать  $n$  ссылочных переменных для обозначения интеграторов:

```
ref (INTGRL) Y1, Y2, ..., YN; (5)
```



и создать объекты класса INTGRL, соответствующие уравнениям системы (1) с помощью операторов вида

YI:—new INTGRL (Y0I,  $\Phi(Y1.Y, Y2.Y, \dots, YN.Y, \text{time})$ ).  
(6)

Замена правой части любого уравнения во время моделирования или добавление нового уравнения, выполняется операторами вида (6), а исключение уравнения — операторами YI.out, исключаяющим интегратор YI из списка INTLIST.

Кроме средств для задания систем обыкновенных дифференциальных уравнений в класс DISCONT входят процедуры, реализующие некоторые возможности активизации процессов по условиям:

1. **procedure** WAITUNTIL (B); **name** B; **boolean** B; ...;

Эта процедура задерживает выполнение текущего процесса до тех пор, пока не станет истинным логическое выражение, заданное ей в качестве фактического параметра.

2. **procedure** ACTCOND (X, B); **name** B;

**ref** (process) X; **boolean** B ...;

Процедура ACTCOND активизирует процесс X в тот момент системного времени, когда становится истинным логическое выражение B.

Для реализации указанных процедур в классе DISCONT предусмотрен список условий (CONDLIST) и два вспомогательных класса (INTERS и WANT). Объекты класса WANT, содержащие ссылки на задержанные процессы, заносятся процедурами WAITUNTIL и ACTCOND в список CONDLIST, а объект класса INTERS после каждого события просматривает этот список, проверяя истинность логических выражений и активизируя в необходимых случаях задержанные процессы.

Следующая программа отображает общую структуру класса DISCONT:

ИМЯ: BDISCO;

**begin**

**simulation class** DISCONT (DT); **real** DT;

**begin ref** (head) CONDLIST, INTLIST;

**ref** (INTEGRATION) INTMET; **ref** (INTERS) CHECK;

**link class** INTGRL; ...;

**process class** INTEGRATION; ...;

**INTEGRATION class** EULIER; ...;

**procedure** WAITUNTIL; ...;

**procedure** ACTCOND; ...;

**link class** WANT; ...;

**process class** INTERS; ...;

```

entry (INTGRL, INTEGRATION, EULIER, WAITUNTIL,
ACTCOND, (YKHD));
INTLIST:—new head; INTMET:—new EULIER;
CONDLIST:—new head; CHECK:—new INTERS; inner;
comment на место inner вставляется операторная часть
программы пользователя, оформленная в виде блока с
префиксом: DISCONT (...) begin ... end;
end DISCONT;
entry (DISCONT, (YKHD));
end

```

Через YKHD в приведенной программе обозначен указатель набора данных, содержащего информацию о классе DISCONT и его атрибутах. В Приложении 9.3 приведена программа класса DISCONT.

В качестве примера использования средств класса DISCONT приведем программу моделирования следующей простой непрерывно-дискретной системы: на некоторый объект, параметры которого ( $Z$  и  $V$ ) изменяются в соответствии с законом, задаваемым системой дифференциальных уравнений

$$\dot{Z} = -0.2Z - 0.5V$$

$$\dot{V} = 0.5Z$$

с начальными условиями

$$Z(0) = 50 \quad V(0) = 0,$$

каждые  $T$  единиц системного времени оказывается дискретное воздействие, вдвое уменьшающее текущее абсолютное значение  $Z$  и  $V$ . Нужно провести моделирование этой системы и определить интервал времени, за который значение  $Z$  станет по модулю меньше EPS. Программа, решающая эту задачу и использующая в качестве входных данных значения  $T$ , EPS и шага интегрирования, имеет вид

Пример 5.5.

```

begin external (YKHD); real T, EPS, ШАГ;
T:=inreal; EPS:=inreal; ШАГ:=inreal;
DISCONT (ШАГ) begin ref (INTGRL) Z, V;
  process class ВОЗДЕЙСТВИЕ;
    begin
      M: Z.Y:=Z.Y/2; V.Y:=V.Y/2; hold (T); goto M;
    end ВОЗДЕЙСТВИЯ;
  Z:—new INTGRL (50, -0.2*Z.Y-0.5*V.Y);
  V:—new INTGRL (0, 0.5*Z.Y);
  activate new ВОЗДЕЙСТВИЕ;

```

```

WAITUNTIL (abs (Z.Y) ≤ EPS);
outtext ('ИСКОМОЕ ВРЕМЯ=');
outfix (time, 3, 10);
end
end

```

Важной особенностью пакетов программ, реализованных в виде библиотек классов и процедур на языке симула-67, является возможность дополнять свойства определенных в библиотеке классов, пользуясь механизмом сочленения. Например, на базе интеграторов, задаваемых в DISCONT декларацией класса INTGRL, в блоке с префиксом DISCONT (...) **begin ... end** можно определить интеграторы, которые можно на некоторое время останавливать путем исключения их из списка интеграторов (после остановки накопленное им значение интеграла будет оставаться неизменным), а затем снова включать в работу. Для этого достаточно описать подкласс класса INTGRL с процедурами STOP и START, выполняющими указанные действия:

```

INTGRL class ONOFINT;
  begin procedure STOP; out;
    procedure START; into (INTLIST);
  end;

```

Имея такое описание, конкретный интегратор L можно задать оператором вида L:— **new ONOFINT (...)**, а его «включение» и «выключение» выполнить соответственно операторами L.START и L.STOP.

### 5.3. Библиотека процедур и классов для построения графиков

Вывод результатов моделирования в виде графиков значительно облегчает качественный анализ работы модели. Разработка программ выдачи графиков на АЦПУ требует значительного объема рутинной работы, связанной с разметкой осей координат, выбором масштаба, выводом нескольких зависимостей на одном графике и т. д. Предлагаемый в данной работе класс SIMTAPL содержит ряд классов и процедур, написанных на языке симула-67 и облегчающих вывод результатов моделирования в форме графиков зависимостей величин от времени или одной величины от другой.

Симула-программа, определяющая класс SIMTAPL, имеет следующую структуру:

```

ИМЯ: BSIMTP;
begin simulation class SIMTAPL;
  begin
    process class PLOT (TITLE, N); text TITLE; integer N;
    begin
      procedure FUNC (Y, C); name Y; real Y; character C; ...;
      procedure PLOTDEP (SF, SA); character SF, SA; ...;
      procedure TABDEP (SF, SA); character SF, SA; ...;
      procedure UPDATE; ...;
      procedure RESET; ...;
      . . . . .
      entry (FUNC, PLOTDEP, TABDEP, UPDATE, RESET,
        (YKHT));
      . . . . .
    end PLOT;
  PLOT class TPLOT (T); real T;
  begin
    procedure OUTPLOT; ...;
    procedure OUTTABLE; ...;
    entry (OUTPLOT, OUTTABLE, (YKHT));
  end TPLOT;
  entry (PLOT, TPLOT, (YKHT));
entry (SIMTAPL, (YKHT));
end

```

Объекты класса PLOT используются при работе программы для хранения (в виде массивов вещественных чисел) значений, соответствующих точкам графика. Параметры TITLE и N, задаваемые при создании объекта класса PLOT, представляющего конкретный график, соответствуют заголовку, печатаемому при выводе графика, и максимальному числу точек графика.

Процедура FUNC, определенная в классе PLOT, используется для задания величин, значения которых будут накапливаться в данном графике. Параметрами этой процедуры являются вещественное выражение Y, задающее величину, которую нужно представить на графике (параметр Y вызывается по наименованию), и литера C, используемая для идентификации этой величины при выдаче графика. Один объект класса PLOT может накапливать значения нескольких величин. Занесение значений всех величин графика, представляющих его очередную точку, выполняется процедурой UPDATE. Процедура RESET уничтожает значения, соответствующие всем точкам графика, освобождая тем самым место для занесения новых точек.

Для вывода на печать графика зависимости величины, идентифицируемой в данном объекте класса PLOT литерой SF,

от величины, идентифицируемой литерой SA, служит процедура PLOTDEP (SF, SA). Процедура TABDEP (SF, SA) выводит зависимость одной величины от другой в виде таблицы.

Объекты класса TPLLOT позволяют удобно задавать построение зависимостей различных величин от времени: каждые T единиц системного времени такой объект снимает значения величин, указанных ему с помощью обращений к процедуре FUNC. Процедура OUTPLOT обеспечивает печать на одном графике зависимостей всех этих величин от времени. Масштаб по времени для всех величин общий, а масштаб по вертикальной оси выбирается для каждой величины так, чтобы обеспечить максимальную точность ее представления. Перед печатью графика выводятся данные о максимальном и минимальном значении каждой величины и выбранном для нее масштабе. Зависимость одной величины от другой может быть выведена с помощью обращения к процедурам PLOTDEP (вывод в виде графика) и TABDEP (вывод в форме таблицы). В Приложении 9.2 приведена программа класса SIMTAPL.

Рассмотрим пример применения средств класса SIMTAPL. Пусть имеется программа моделирования некоторой обслуживающей системы, в которой переменная M содержит текущее значение интенсивности обслуживания, а переменная L — интенсивность входного потока требований. M и L меняются с течением времени. В переменных ТОЖ и ТРАБ модель накапливает значения суммарного времени, затраченного требованиями на ожидание обслуживания, и суммарного времени работы обслуживающего прибора.

Для построения графика зависимости величин M, L, ТОЖ и ТРАБ от времени, нужно создать объект класса TPLLOT, указав в качестве фактических параметров требуемое количество точек и периодичность снимка значений этих величин:

```
ref (TPLLOT) ГРАФ;  
ГРАФ:—new TPLLOT ('M, L, ТОЖ, ТРАБ  
КАК ФУНКЦИИ ВРЕМЕНИ', 100, 0.5);
```

и указать посредством процедуры FUNC объекту ГРАФ те величины, которые он должен регистрировать и соответствующие им символы для представления их на печати:

```
ГРАФ.FUNC (M, "M"); ГРАФ.FUNC (L, "L");  
ГРАФ.FUNC (ТОЖ, "*"); ГРАФ.FUNC (ТРАБ, "P");
```

По истечении определенного промежутка времени можно вывести накопленную в графике ГРАФ информацию в графической форме с помощью оператора ГРАФ.OUTPLOT или в форме таблицы: ГРАФ.OUTTABLE.

С помощью оператора ГРАФ.PLOTDER ("\*", "M") можно получить график зависимости ТОЖ от M.

Чтобы вновь начать накопление информации о наблюдаемых значениях величин M, L, ТОЖ, ТРАБ, нужно уничтожить старую информацию о них, употребив оператор ГРАФ.RESET.

Для обеспечения возможности доступа к средствам класса SIMTAPL в первом блоке программы должно быть употреблено описание **external** (УКНТ), и конструкция **simulation begin** заменена на **SIMTAPL begin**.

Если в одной программе необходимо одновременно пользоваться средствами класса DISCONT (см. 5.2) и средствами построения графиков (класс SIMTAPL), то можно дополнить один из них, например, DISCONT средствами другого с помощью программы

```
ИМЯ: BDISCP;  
begin external (УКНТ);  
SIMTAPL class DISCONT (DT); real DT;  
  begin <атрибуты класса DISCONT>;  
    entry (... , (УКНТD));  
  end;  
entry (DISCONT, (УКНТD));  
end
```

Выполнив трансляцию данной программы, можно пользоваться средствами обоих классов. В качестве примера такого использования рассмотрим построение графиков зависимостей параметров объекта Z и V из системы, представляемой программой примера 5.5, от времени и поведения величины Z как функции V. Для этого нужно добавить в программу 5.5 еще одно описание с **external**:

```
ИМЯ: GRAF;  
begin external (УКНТ); external (УКНТD); real T, EPS,  
ШАГ;
```

а в блок с префиксом DISCONT (ШАГ) **begin** добавить описания и операторы, задающие объект класса TPLOT:

```
ref (TPLOT) ГРАФИК;  
ГРАФИК:—new TPLOT ('Z=F(T), V=F(T).', 100, 10*ШАГ);  
ГРАФИК.FUNC (Z,Y, "Z"), ГРАФИК.FUNC (V,Y, "V");
```

Для выдачи графика на печать можно использовать оператор ГРАФИК.OUTPLOT, который выдает на одном графике зависимости Z и V от времени. Оператор ГРАФИК.PLOTDER ("Z", "V") печатает зависимость Z от V.

В приложении 9.4 приведена программа примера 5.5, дополненная средствами построения графиков, и результаты ее работы. Трансляция и счет проводились на ЕС ЭВМ.

## 5.4. Пакет программ для сбора статистики

В данном разделе рассмотрим структуру и основные средства пакета программ для сбора и вывода статистических данных о работе имитационных моделей. Этот пакет описан в работе [38] и представляет собой набор классов и процедур на языке симула-67, который разработан в Исследовательской лаборатории ВМС США и применялся там в программах моделирования систем спутниковой связи. В ИПМ им. М. В. Келдыша АН СССР программы пакета были адаптированы для работы на ЭВМ БЭСМ-6 и ЕС ЭВМ и использовались в имитационной модели ЭВМ CRAY-1 [4].

Средства пакета SIMSTAT позволяют накапливать и выводить на печать в наглядной форме следующие статистические данные об арифметических величинах, характеризующих работу модели:

- 1) сумму наблюдаемых значений;
- 2) количество наблюдений;
- 3) времена первого и последнего наблюдений;
- 4) максимальное и минимальное наблюдаемые значения;
- 5) среднее значение величины и ее дисперсию, вычисляемые как на основе одинакового веса всех наблюдений, так и с взвешиванием каждого наблюдаемого значения по времени, в течение которого это значение не изменялось;
- 6) три типа гистограмм, характеризующих распределение величины: а) с равным весом всех наблюдений, б) с взвешиванием наблюдаемого значения временем, в течение которого оно не менялось, в) с табуляцией времени наблюдения, взвешенного значением наблюдаемой величины.

Для сбора статистики с каждой величиной, поведение которой исследуется, связывается объект, принадлежащий одному из классов, определенных в SIMSTAT. Атрибуты этого объекта используются для накопления статистических данных, характеризующих соответствующую величину. Обновление накопленных данных и выдача статистики производится с помощью специальных процедур, которые также являются атрибутами указанного объекта. Обращения к этим процедурам задаются пользователем в тех местах программы, где происходят изменения исследуемой величины, или когда нужно напечатать накопленные данные.

Программа, содержащая пакет SIMSTAT и обеспечивающая запись его в библиотеку, имеет следующую структуру:

ИМЯ: BSTAT;

**begin**

**simulation class SIMSTAT;**

**begin** *<описание глобальных переменных, используемых процедурами и классами, определенными в SIMSTAT>;*

**link class STAT(VNAME, VDECC); value VNAME, VDECC;**  
**text VNAME, VDECC;**

**begin.....**

**procedure FULL REPORT; ...;**

**entry (FULL REPORT, (YKHS));**

**end STAT;**

**STAT class SINTOPS;**

**begin.....**

**procedure REPORT; ...;**

**entry (REPORT, (YKHS));**

**end SINTOPS;**

**SINTOPS class STATHIST (NBINS, HTYPE, LOWERBD, INC);**

**integer NBINS, HTYPE, LOWERBD, INC;**

**begin...**

**procedure UPDATE (NEWVAL, TOBS); real NEWVAL, TOBS; ...;**

**procedure OUTHIST; ...;**

**entry (UPDATE, OUTHIST, (YKHS));**

**end STATHIST;**

**SINTOPS class STATREAL;**

**begin procedure UPDT(NEWVAL, TOBS); ...;**

**...**

**entry (UPDT, (YKHS));**

**end STATREAL;**

**...**

**entry (STAT, SINTOPS, STATHIST, STATREAL, (YKHS));**

**end SIMSTAT;**

**entry (SIMSTAT, (YKHS));**

**end**

где YKHS — указатель набора данных, в который будет занесена необходимая информация о классе SIMSTAT.

**З а м е ч а н и е.** В данной программе представлены заголовки доступных пользователю классов и процедур и структура иерархии классов, описывающих статистические объекты.

Назначение процедур и классов, определенных в SIMSTAT, рассмотрим на примерах их использования для сбора статистики. Пусть в некоторой программе моделирования нас интере-



суют статистические характеристики длины очереди, отображаемой множеством (набором) **K**, которое задано с помощью описания **ref (head) K** и инициализировано оператором **K:— new head**. Для накопления статистики о количестве элементов в наборе **K** в виде гистограммы с равным весом каждого наблюдения нужно перед началом моделирования создать объект класса **STATHIST**, задав его имя и соответствующие фактические параметры, служащие для идентификации данных при выводе (**VNAME**, **VDECC**) и определяющие тип гистограммы (**HTYPE**), количество интервалов (**NBINS**), правую границу первого интервала (**LOWERBD**) и ширину интервала (**INC**). Это можно сделать, описав ссылочную переменную **ref (STATHIST) HISTK** и присвоив ей значение оператором **HISTK:— new STATHIST ('HISTK', 'ГИСТОГРАММА ДЛИНЫ K', 8, 1, 0, 4)**, который задаст гистограмму с 8 интервалами (**NBINS=8**), все наблюдения будут иметь одинаковый вес, равный 1 (**HTYPE=1**), правая граница первого (бесконечного) интервала гистограммы будет равна 0 (**LOWERBD = 0**), а ширина следующих интервалов — 4 (**INC = 4**).

**З а м е ч а н и е.** Для определения гистограммы, в которой каждое наблюдение взвешивается временем пребывания соответствующего значения, нужно задать параметр **HTYPE=2**, а для гистограммы, показывающей распределение моментов времени, в которые происходило изменение величины, взвешенных соответствующими этим моментам значениями,— **HTYPE=3**.

Обновление гистограммы **HISTK** выполняется процедурой **UPDATE**, обращения к которой должны происходить при изменении количества объектов, присутствующих в наборе **K**, т. е. в тех точках программы, где происходит включение или исключение объектов в множество **K**, должен стоять оператор **HISTK.UPDATE (K. cardinal, time)**. Печать гистограммы **HISTK** обеспечивается оператором **HISTK.OUTHIST**.

Пользуясь сбором статистики в виде гистограмм, следует учитывать, что затраты времени на их обновление и расход памяти на хранение значений, пропорциональны количеству интервалов гистограммы.

В тех случаях, когда нет необходимости в получении гистограммы распределения некоторой величины, можно обеспечить вычисление ее основных статистических характеристик, перечисленных в пунктах 1—5 в начале раздела. Для этого с исследуемой величиной необходимо связать статистический объект класса **STATREAL**. Например, описание **ref (STATREAL) СТАТК** и оператор

**СТАТК: — new STATREAL ('СТАТК', 'ДЛИНА ОЧЕРЕДИ K'),**

задают статистический объект для хранения, вычисления и вывода основных характеристик очереди, отображаемой набором K. Операторы СТАТК. UPDT (K. cardinal, time), исполняемые в те моменты времени, когда происходит изменение длины очереди, обеспечивают обновление статистических данных.

Вывод статистики, накопленной в объекте СТАТК, производится оператором СТАТК. REPORT, который печатает названия основных характеристик и их значения вместе с идентифицирующей информацией, заданной при генерации объекта СТАТК.

Если необходимо вывести информацию, накопленную во всех использованных в программе статистических объектах классов STATREAL и STATHIST, то это можно сделать при помощи обращения к процедуре FULL REPORT посредством оператора СТАТК.FULL REPORT, который обеспечит вывод всей статистики в той же форме, что и процедуры OUTHIST и REPORT.

## Приложение 1

### ОТЛИЧИЯ ЯЗЫКА СИМУЛА-67 ОТ АЛГОЛА-60

1. По умолчанию параметры передаются не по наименованию, как в алголе-60, а по значению и по ссылке.

2. Все переменные получают начальные значения в момент входа в блок, в котором эта переменная определяется.

ТИП	НАЧАЛЬНОЕ ЗНАЧЕНИЕ
-----	--------------------

<b>real, integer</b>	<b>0</b>
<b>ref</b>	<b>none</b>
<b>boolean</b>	<b>false</b>
<b>text</b>	<b>notext</b>

3. Понятие собственных (**own**) переменных алгола отсутствует в симула-67.

4. Алгольный тип «строка» заменяется на понятие **text**.

5. Симула-67 содержит расширения к алголу-60: вызов параметров по ссылке, понятие класса и ссылочной переменной, средства построения иерархий деклараций классов, сопрограммы (**resume** и **detach**), стандартизованные средства ввода-вывода.

## Приложение 2

### ПРАВИЛА ПЕРЕДАЧИ ПАРАМЕТРОВ

Рассмотрим влияние трех способов передачи параметров, называемых вызовами по значению, по наименованию и по ссылке.

**Вызов по значению.** Формальный параметр, вызываемый по значению, имеет много общих свойств с локальными величинами, описанными в теле процедуры. Отличие состоит в том, что в то время, как локальная величина получает первоначальное значение в соответствии со стандартными соглашениями симула-67, формальный параметр, вызываемый по значению, получает начальное значение в результате вычисления фактического параметра в момент вызова.

**Пример 2.1.** Программный сегмент

**procedure P1(K); integer K;**

```

begin integer J;
  for J:=1 step 1 until K do...
end OF P1;

```

. . .  
 P1(5);

можно рассмотреть как

```

procedure P1;
  begin integer K, J;
    K:=5;
    . . .
    for J:=1 step 1 until K do...
  end OF P1;

```

. . .  
 P1;

Вызов по наименованию. Параметр, вызываемый по наименованию, никогда не вычисляется во время вызова процедуры. Вместо этого, когда в теле процедуры встречается параметр, вызываемый по наименованию, то соответствующий фактический параметр (который может быть выражением) пере-вычисляется и результат вычисления используется вместо данного вхождения формального параметра. Такая трактовка позволяет рассматривать каждое появление в теле процедуры формального параметра, вызываемого по наименованию, как текстуальную замену на соответствующий фактический параметр. (Возможные конфликты имен, возникающие при такой модификации программы, систематически разрешаются компилятором.)

Пример 2.2. Рассмотрим сегмент

```

array A [1:5]; integer I;
procedure P2(P); name P; real P;
  begin real X;
    P:=...; X:=P...;
  end OF P2;
  . . .
  P2(A [I]);

```

Эквивалентный сегмент, не использующий параметр, вызываемый по наименованию, приводится ниже:

```

array A [1:5]; integer I;
procedure P2;
  begin real X;
    A [I]:=...; X:=A [I]...;
  end OF P2;
  . . .
P2;

```

Вызов по ссылке. Вызов по ссылке, в известном смысле, представляет компромисс между вызовом по наименованию и вызовом по значению. Этот вызов применим для некоторых типов параметров, а именно для тех, которые ссылаются на структуры данных или программные компоненты. Основная идея заключается в том, что соответствующий фактический параметр вычисляется во время вызова, и результат этого вычисления, который должен быть ссылкой (на структуру данных или программную компоненту), затем используется в тех местах, где встречается этот формальный параметр.

Пример 2.3. Рассмотрим сегмент

```
text S;
procedure P3(T); text T;
comment подразумевается вызов по ссылке;
begin...
    outtext (T);
    . . .
end OF P3;
. . .
P3(S);
```

Эта последовательность ведет себя так же, как и последовательность

```
text S;
procedure P3;
begin text T; T:=S; ...
    outtext (T); ...
end OF P3;
. . .
P3;
```

Способы передачи параметров для процедур даны в таблице.

Т а б л и ц а

Параметр	Способы передачи		
	по значению	по ссылке	по наименованию
<обыкновенный тип>	В	НР	Р
<тип ссылок на объект>	НР	В	Р
text	Р	В	Р
<обыкновенный тип> array	Р	В	Р
<ссылочный тип> array	Р	В	Р
<тип> procedure	НР	В	Р
label	НР	В	Р
switch	НР	В	Р

**З а м е ч а н и е.** Выделенная рамкой часть таблицы соответствует передачам параметров деклараций классов. Обозначения в таблице: В — выполняется при отсутствии указания о виде передачи, Р — разрешается при наличии указания о виде передачи, НР — не разрешается.

### П р и л о ж е н и е 3

#### СООБЩЕНИЯ О ДИНАМИЧЕСКИХ ОШИБКАХ (ЕС-ЭВМ)

Приведем тексты распечаток об ошибках и причины их возникновения:

##### SML101 ЗНАЧЕНИЕ ТЕКСТОВОЙ ССЫЛКИ=NOTEXT

При вычислении конструкции вида  $T.F(\dots)$ , где  $F$  — одна из процедур-атрибутов текста, значение выражения  $T$  оказалось равным `notext` (если, конечно,  $F$  не определена и при  $T=\text{notext}$ ).

##### SML102 УКАЗАТЕЛЬ ПОЗИЦИИ ВНЕ ИНТЕРВАЛА [1, T. LENGTH]

При попытке обращения к литерам текста, например, с помощью конструкции `T.getchar`, `T.putchar(...)` значение указателя позиции текста оказалось вне допустимого интервала.

##### SML103 ПЕРЕХОД НА НЕАКТИВНУЮ КОМПОНЕНТУ

При выполнении оператора `goto` производится попытка перехода на неактивный объект.

##### SML104 ИСЧЕРПАНА ПАМЯТЬ В ПОЛЕ ДАННЫХ

При попытке отвести в поле данных память во время порождения объекта, вызова процедуры и т. п. не удалось выделить требуемое количество памяти служебными программами транслятора. В этом случае необходимо повторить запуск симула-программы на счет (трансляцию программы можно не повторять, если она была записана в постоянный набор данных) в большем разделе (при работе в режиме MFT) или с увеличенным параметром REGION (в режиме MVT).

##### SML107 НЕВЕРНЫЕ ЗНАЧЕНИЯ ПАРАМЕТРОВ

Неверно заданы параметры текстовой процедуры `sub(K, N)`, т. е. не выполнены условия  $K \geq 1 \wedge N \geq 0 \wedge K + N \leq \text{length} + 1$ .

##### SML108 ТЕКСТ УКРОЧЕН НА: <целое>

Длина текста (L), в который производится запись, недостаточна, чтобы вместить выводимый текст или результат редактирования числа.

Во вводимый текст переписывается L—1 литера вводимого текста. В последнюю литеру вставляется символ ? (для БЭСМ-6 знак ↑). Значение <целое> равно разности между длиной выводимого текста и значением L.

#### SML109 НИЖНЯЯ ГРАНИЦА МАССИВА <имя массива> БОЛЬШЕ ВЕРХНЕЙ

Ошибка в задании граничных пар при описании массива. Отведение памяти под массив не производится. В последующем возможно появление наведенных ошибок (номер SML112).

#### SML110 НЕСООТВЕТСТВИЕ ФАКТИЧЕСКОГО И ФОРМАЛЬНОГО ПАРАМЕТРА

Тип или вид фактического параметра процедуры не совпадает с типом или видом формального параметра.

#### SML111 КОЛИЧЕСТВО ФАКТИЧЕСКИХ И ФОРМАЛЬНЫХ ПАРАМЕТРОВ НЕ СОВПАДАЕТ

Количество передаваемых параметров не соответствует количеству параметров в описании процедуры.

З а м е ч а н и е (к SML110 и SML111). Во время выполнения программы контроль соответствия формальных и фактических параметров производится лишь при вызове формальных и виртуальных процедур. Под термином формальная процедура подразумевается формальный параметр со спецификатором **procedure** или <тип> **procedure**. В остальных случаях контроль соответствия формальных и фактических параметров производится при трансляции.

#### SML112 НЕ СОВПАДАЕТ РАЗМЕРНОСТЬ МАССИВА <имя массива>

Число индексов у переменной не совпадает с числом граничных пар в описании массива.

#### SML113 ИНДЕКС ВЫХОДИТ ЗА ГРАНИЦЫ МАССИВА <имя массива>

Индекс у переменной с индексами выходит за границы, определенные в описании массива.

#### SML114 ДЛИНА МАССИВА <имя массива> БОЛЬШЕ ОБЪЯВЛЕННОЙ В COMMON-БЛОКЕ

Длина массива, описанного в программе, больше, чем длина, определенная его описанием в общем блоке.

## SML115 ЗНАЧЕНИЕ ПАРАМЕТРА $< 0$

В процедурах редактирования `putfix (I, N)`, `putreal (R, N)`, `putfrac (I, N)`, значение параметра  $N < 0$ . Редактирование не выполняется.

## SML116 ЗНАЧЕНИЕ ОПЕРАЦИИ ВОЗВЕДЕНИЯ В СТЕПЕНЬ НЕ ОПРЕДЕЛЕНО

Некорректное значение операндов операции возведения в степень. Результат операции не определен.

## SML117 ЗНАЧЕНИЕ ЛЕВОЙ ЧАСТИ = NOTEXT

При присваивании текстовых значений значение левой части равно `notext`, а правой не равно `notext`.

## SML118 ЗНАЧЕНИЕ ПАРАМЕТРА $N > 11$

В компиляторе введено ограничение на переменную  $N$  функции редактирования `putfrac (I, N)`. Значение  $N$  должно быть меньше или равно 11.

## SML119 ЗАПИСЬ ЧИСЛА НЕ НАЙДЕНА

Текст, обрабатываемый одной из процедур доредактирования, не содержит записи числа.

## SML120 ЗНАЧЕНИЕ ПАРАМЕТРА ВНЕ ДИАПАЗОНА

Значение фактического параметра при обращении к процедуре `CHAR` выходит за интервал  $[64, 254]$ .

## SML121 АРГУМЕНТ EVTIME НЕ ПРЕДСТАВЛЕН В УС

При выполнении конструкции `X.evtime` объектное выражение `X` указывает на пассивный или завершенный процесс.

## SML122 ФАЙЛ УЖЕ ОТКРЫТ

Повторное открытие файла ввода-вывода. Оператор игнорируется.

## SML123 ДЛИНА ПЕЧАТНОЙ СТРОКИ $> 127$ СИМВОЛОВ

При открытии файла, связанного с объектом класса `printfile`, длина буфера вывода `image` больше 127 символов.

## SML124 ФАЙЛ УЖЕ ЗАКРЫТ

Повторное закрытие файла ввода-вывода. Оператор игнорируется.

## SML125 КОНЕЦ ФАЙЛА



Попытка чтения с файла ввода после того, как встретился признак конца файла.

### SML126 ЗАПИСЬ ЧИСЛА ПРЕВЫШАЕТ МАКСИМАЛЬНО ДОПУСТИМОЕ ЗНАЧЕНИЕ

При выполнении процедур дерадктирования полученное значение числа превышает максимально представимое. В качестве результата процедуры выдается максимальное положительное число требуемого вида.

### SML127 ВВОДИМЫЙ ТЕКСТ НЕ ПОМЕЩАЕТСЯ В БУФЕР (IMAGE)

В процедуре ввода  $\text{intext}(T)$  значение переменной  $T$  больше длины буфера, определенной при открытии файла ввода.

### SML128 ДЛИНА ВВОДИМОГО ТЕКСТА $< 0$

В процедуре ввода  $\text{intext}(K)$  значение параметра  $K < 0$ .

### SML130 ДЛИНА ВЫВОДИМОГО ТЕКСТА $< 0$

Параметр  $L$ , определяющий длину подтекста, с которым работают процедуры вывода числовых значений ( $\text{outint}(K, L)$ ,  $\text{outfix}(A, N, L)$ ,  $\text{outreal}(A, N, L)$ ,  $\text{outfrac}(K, N, L)$ ), имеет отрицательное или нулевое значение.

### SML131 ВЫВОДИМЫЙ ТЕКСТ НЕ ПОМЕЩАЕТСЯ В БУФЕР (IMAGE)

Длина текста, выводимого процедурами класса  $\text{outfile}$ , превышает длину буфера, определенную при открытии файла вывода.

### SML132 ИМЯ ФАЙЛА = NOTEXT

Значение  $\text{ref}$ -переменной, которая должна обозначать объект класса ввода-вывода, равно  $\text{none}$ .

### SML133 ФАЙЛ ЗАКРЫТ

Попытка чтения или записи информации с закрытого файла с помощью процедур ввода-вывода.

### SML134 НЕДОПУСТИМЫЙ ПАРАМЕТР $N$ В ПРОЦЕДУРЕ $\text{EJECT}$

Значение параметра  $N$  процедуры  $\text{eject}(N)$  должно лежать в пределах  $1 \leq N \leq \text{lines per page}$ .

### SML135 НЕЗАКОННОЕ ПРИСВАИВАНИЕ

При выполнении присваивания ссылок значением правой части является объект, не принадлежащий к классу, квалифицирующему левую часть, или к одному из его подклассов.

#### SML136 НЕДОПУСТИМЫЙ ОБЪЕКТ В QUA

При вычислении конструкции **X qua C** значением выражения **X** является объект класса, не содержащегося в **C**.

#### SML137 НЕДОПУСТИМЫЙ ОБЪЕКТ В THIS

Конструкция **this C** употреблена вне тела класса **C** или его подкласса. Это же сообщение выдается, если **this C** вычисляется в теле класса **C**, употребленного в качестве префикса к блоку.

#### SML138 НЕВЕРНОЕ УПОТРЕБЛЕНИЕ DETACH

Оператор **detach** употреблен вне тела класса или квазипараллельной системы.

#### SML139 АРГУМЕНТ RESUME НЕСАМОСТОЯТЕЛЬНЫЙ ОБЪЕКТ

При выполнении оператора **resume(X)** значением выражения **X** является прикрепленный или завершенный объект. Оператор не выполняется.

#### SML140 АРГУМЕНТ RESUME — NONE

Значение выражения **X** в операторе **resume(X)** равно **none**. Оператор не выполняется.

#### SML141 ЗНАЧЕНИЕ ОБЪЕКТНОГО ВЫРАЖЕНИЯ = NONE

Значение объектного выражения слева от точки в дистанционном идентификаторе равно **none**. Значение дистанционного идентификатора не определено, возможны наведенные ошибки. В счетном режиме проверка объектного выражения на **none** не производится.

#### SML142 НЕВЕРНО ЗАДАН РАЗМЕР БУФЕРА ВВОДА ПЕРФОКАРТ

При открытии файла ввода с перфокарт длина буфера ввода отлична от 80.

#### SML143 "ХОЛОСТАЯ" ВИРТУАЛЬНАЯ

Для идентификатора, упомянутого в списке виртуальных величин некоторого класса, нет описания в блоке тела этого класса или его подклассов.

Тексты сообщений об ошибках, возникающих из-за некорректного задания фактических параметров встроенных арифме-

тических функций не приводятся, так как они исчерпывающе характеризуют причины ошибок.

**З а м е ч а н и е.** Тексты сообщений о динамических ошибках для транслятора на БЭСМ-6, могут незначительно отличаться от приведенных выше.

#### Приложение 4

### СООБЩЕНИЯ КОМПИЛЯТОРОВ

(тексты приведены для компилятора на ЕС ЭВМ)

#### SML1 НЕКОРРЕКТНОЕ УПОТРЕБЛЕНИЕ ВЫРАЖЕНИЯ ТИПА <тип>

Это сообщение выдается тогда, когда выражение указанного типа нельзя употреблять в данном контексте. Например, булевское выражение не может использоваться в качестве индекса или граничной пары массива. Чаще всего сообщение выдается при ошибке в задании фактических параметров для стандартных процедур, если, например, в качестве параметра задано выражение неподходящего типа.

#### SML2 НЕДОПУСТИМОЕ ОТНОШЕНИЕ ТЕКСТОВ

#### SML3 НЕЦЕЛЫЕ АРГУМЕНТЫ ОПЕРАЦИИ

#### ЦЕЛОЧИСЛЕННОГО ДЕЛЕНИЯ

#### SML4 В ВЫРАЖЕНИИ НЕСОВМЕСТИМЫЕ ТИПЫ

#### ОПЕРАНДОВ

#### SML5 В УСЛОВНОМ ВЫРАЖЕНИИ НЕСОВМЕСТИМЫЕ

#### ТИПЫ ПОДВЫРАЖЕНИЙ

#### SML6 НЕТ ELSE В УСЛОВНОМ ВЫРАЖЕНИИ

#### SML7 ОТСУТСТВУЕТ ВЫРАЖЕНИЕ

Сообщение выдается в случае, если пропущено выражение, а соответствующая языковая конструкция не допускает его отсутствия. Например, пропущено булевское выражение в конструкции **if... then** или объектное выражение в операторе **activate**.

#### SML8 НЕКОРРЕКТНАЯ ЛЕВАЯ ЧАСТЬ

#### ПРИСВАИВАНИЯ

#### SML9 КЛАССЫ НЕ ПРИНАДЛЕЖАТ ОДНОЙ ЦЕПОЧКЕ

#### ПРЕФИКСОВ: <идентификатор 1>

#### <идентификатор 2>

Сообщение сигнализирует об ошибке в употреблении имен классов в конструкции **X qua A**. Идентификатор класса **A** должен находиться в одной цепочке префиксов с классом, служащим квалификацией выражения **X**.

#### SML10 НЕСООТВЕТСТВИЕ ТИПОВ В ОПЕРАТОРЕ ПРИСВАИВАНИЯ

## SML11 ОШИБКА В ПРИСВАИВАНИИ ССЫЛОК

Конструкция, стоящая в левой части объектного присваивания, имеет тип, отличный от типа «ссылка на объект».

## SML12 ВЫРАЖЕНИЕ НЕ ИМЕЕТ КВАЛИФИКАЦИИ

На месте объектного выражения, например в дистанционном идентификаторе или планирующем операторе, употреблено выражение другого типа.

## SML13 ПОСЛЕ NEW НЕТ ИМЕНИ КЛАССА

## SML14 ОШИБКА В КВАЛИФИКАЦИИ

На месте объектного выражения использовано выражение другого типа.

## SML15 ОШИБКА В ДИСТАНЦИОННОМ ИДЕНТИФИКАТОРЕ

## SML16 ОТСУТСТВУЕТ КЛАСС, ОХВАТЫВАЮЩИЙ КВАЛИФИКАЦИИ ОБЕИХ АЛЬТЕРНАТИВ

## SML17 ОТСУТСТВУЕТ ИМЯ КЛАССА ПОСЛЕ СИМВОЛА QUA

## SML18 ОТСУТСТВУЕТ ИМЯ КЛАССА ПОСЛЕ СИМВОЛА THIS

## SML19 НЕКОРРЕКТНАЯ ЗАПИСЬ ЭЛЕМЕНТА МАССИВА

## SML20 ОШИБКА В КОМПИЛЯТОРЕ

Компилятор не смог распознать и оттранслировать какую-то часть указанного оператора. Проверьте синтаксическую правильность его записи и исправьте обнаруженные ошибки, а если их нет — передайте листинг разработчикам компилятора.

## SML21 НЕВЕРНЫЙ ОПЕРАНД ВЫРАЖЕНИЯ

## SML22 ВЛОЖЕННОСТЬ БЛОКОВ БОЛЬШЕ 15

## SML23 НЕДОПУСТИМЫЙ ТИП ПОКАЗАТЕЛЯ СТЕПЕНИ <тип>

## SML24 НЕДОПУСТИМЫЙ ТИП ОСНОВАНИЯ СТЕПЕНИ <тип>

## SML25 ОШИБКА В ВЫРАЖЕНИИ

## SML26 ПЕРЕМЕННАЯ НЕ ССЫЛОЧНОГО ТИПА: <идентификатор>

Указанная в сообщении переменная употреблена в контексте, требующем, чтобы она имела ссылочный тип, например, переменная X в дистанционном идентификаторе X. А должна иметь тип «ссылка на объект».

## SML27 АТРИБУТ ТЕКСТА НЕ ПРОЦЕДУРА (ПРОЦЕДУРА-ФУНКЦИЯ)

В конструкции вида T.F, где T — текстовое выражение, идентификатор F оказался отличным от идентификаторов процедур, являющихся атрибутами текста.

- SML28 НЕДОПУСТИМОЕ ВЫРАЖЕНИЕ В  
ДИСТАНЦИОННОМ ИДЕНТИФИКАТОРЕ
- SML29 НЕДОПУСТИМАЯ ОПЕРАЦИЯ НАД  
ДИСТАНЦИОННЫМ ИДЕНТИФИКАТОРОМ
- SML30 ОШИБКА В ЗАПИСИ ЦИКЛА
- SML31 ОШИБКА В ОПЕРАТОРЕ ENTRY

Сообщение выдается, если среди аргументов оператора entry имеются такие, которые не являются идентификаторами классов или процедур, или не имеют вид  $E=A$  или  $E(A)$ , где E — идентификатор процедуры (класса), а A — его внешнее имя. Последний аргумент оператора entry должен иметь вид (H), где H — указатель набора данных (на ЕС ЭВМ — DD-предложение, на БЭСМ-6 — номер направления и номер зоны, более подробно см. 4.3.1). Запись информации в набор данных не производится, что может повлечь ошибки в программе, использующей в качестве внешних те имена, которые перечислены в entry.

- SML32 ПОВТОРЕНИЕ ИМЕНИ В СОВОКУПНОСТИ  
ВИРТУАЛЬНЫХ: <идентификатор>
- SML33 ПРОТИВОРЕЧИВОЕ ИСПОЛЬЗОВАНИЕ ИМЕНИ  
В ОПИСАНИЯХ И В СОВОКУПНОСТИ  
ВИРТУАЛЬНЫХ: <идентификатор>
- SML34 ЛИШНЯЯ ЗАПЯТАЯ
- SML35 ОШИБКА В СОВОКУПНОСТИ СПЕЦИФИКАЦИЙ  
ИЛИ ЗНАЧЕНИЙ
- SML36 ОШИБКА В EXTERNAL

Элементы списка в описании, начинающемся с символа external, имеют тот же синтаксис, что и аргументы оператора entry (см. 4.3.2, и комментарии к сообщению SML31). Указанное сообщение выдается в случае нарушения синтаксиса хотя бы в одном элементе списка. Чтение информации с набора данных не производится, в связи с чем возможна наведенная диагностика.

#### SML37 ИСПОРЧЕНА ИНФОРМАЦИЯ

Компилятор обнаружил искажение информации о внешних классах и/или процедурах, записанной в набор данных, либо ее отсутствие. Причиной искажения или отсутствия информации может быть ошибка в операторе entry (см. SML31), сбой внешних устройств, ошибка в задании указателя набора дан-

ных в операторе **entry** или в описании **external**, физическая порча магнитного носителя, запись в указанный набор данных другой информации. Возможна наведенная диагностика.

SML38 НЕ ЗАКОНЧЕН БЛОК, КЛАСС ИЛИ  
ПРОЦЕДУРА. ВСТАВЛЕН **END**  
SML39 ТЕКСТ ВНЕ ПРОГРАММЫ

Сообщение выдается в том случае, если компилятор обнаруживает операторы или описания вне самого внешнего блока программы, где может находиться лишь конструкция, задающая имя программы: **ИМЯ**: <идентификатор>;, которая должна помещаться в программе перед первым символом **begin**. На ЕС ЭВМ вслед за идентификатором, обозначающем имя программы, может быть задан режим трансляции. Чаще всего причиной выдачи указанного сообщения служит отсутствие символа ; (точки с запятой) после имени программы.

SML40 ЛИШНИЙ **END**

Сообщение выдается в случае нарушения баланса символов **begin** и **end**. Если их количество совпадает, то необходимо проверить, не оказался ли какой-либо символ **begin** в позиции комментария или не пропущена ли где-нибудь точка с запятой, завершающая описание процедуры или декларацию класса.

SML41 НЕВЕРНАЯ ЗАПИСЬ ВНЕШНЕЙ ПРОЦЕДУРЫ

Допущена ошибка в операторе **code**, задающем имя внешней процедуры и название языка, на котором она написана (см. 4.4.1).

SML42 ОШИБКА В ОПИСАНИЯХ

Чаще всего причиной выдачи сообщения является пропуск символа ; (точки с запятой), отделяющего одно описание от другого.

SML43 ОШИБКА В ОПИСАНИИ МАССИВА

SML44 НЕВЕРНЫЕ ГРАНИЧНЫЕ ПАРЫ МАССИВА

SML45 НЕВЕРНЫЙ ОПЕРАТОР

В позиции оператора употреблена конструкция, которая им не является. Чаще всего следствие ошибки в набивке.

SML46 ПЕРЕХОД НЕ ПО МЕТКЕ <идентификатор>

SML47 НЕДОПУСТИМЫЙ ЭЛЕМЕНТ В ОПЕРАТОРЕ

ВВОДА: <идентификатор>

SML48 ОШИБКА В ПАРАМЕТРАХ СТАНДАРТНОЙ  
ПРОЦЕДУРЫ

Сообщение выдается при несоответствии типов и/или числа формальных параметров стандартных (системных) процедур, определенных в языке или в системных классах.

**SML49 НЕКОРРЕКТНОЕ ИСПОЛЬЗОВАНИЕ ИМЕНИ**

⟨идентификатор⟩

**SML50 ОШИБОЧНОЕ ОБРАЩЕНИЕ К АТТРИБУТУ**

КЛАССА: ⟨идентификатор 1⟩⟨идентификатор 2⟩

В тексте симула-программы имеется обращение к атрибуту ⟨идентификатор 2⟩ класса ⟨идентификатор 1⟩, не описанному в данном классе.

**SML51 НЕ ОПРЕДЕЛЕН КЛАСС ⟨идентификатор⟩**

**SML52 ОШИБКА В ПРИСОЕДИНЕНИИ**

Ошибка в синтаксисе присоединяющего оператора. Сообщение выдается также в том случае, если после символа **when** стоит не идентификатор класса.

**SML53 НЕТ СПЕЦИФИКАЦИИ ДЛЯ ⟨идентификатор⟩**

**SML54 ОШИБКА В СПИСКЕ ФОРМАЛЬНЫХ  
ПАРАМЕТРОВ**

**SML55 НЕДОПУСТИМЫЙ СПОСОБ ВЫЗОВА ПАРАМЕТРА  
⟨идентификатор⟩**

**SML56 НЕСООТВЕТСТВИЕ ЧИСЛА ФОРМАЛЬНЫХ И  
ФАКТИЧЕСКИХ ПАРАМЕТРОВ**

**SML57 НЕВЕРНЫЙ ФАКТИЧЕСКИЙ ПАРАМЕТР ДЛЯ  
ФОРМАЛЬНОГО ПАРАМЕТРА ⟨идентификатор⟩**

**SML58 НЕДОПУСТИМЫЙ ТИП ВЫРАЖЕНИЯ В  
ПЛАНИРУЮЩЕМ ОПЕРАТОРЕ**

**SML59 ПЛАНИРУЕТСЯ НЕ ПРОЦЕСС ⟨идентификатор⟩**

Аргументом планирующего оператора служит объектное выражение, обозначающее объект, который не принадлежит ни к одному из подклассов класса **process** (см. § 2.1).

**SML60 НЕВЕРНОЕ УПОТРЕБЛЕНИЕ PRIOR**

**SML61 НЕВЕРНО ЗАДАНЫ ГРАНИЦЫ МАССИВА ИЗ  
COMMON-БЛОКА**

**SML62 ОШИБКА В ИМЕНИ COMMON-БЛОКА**

**SML63 ПРОТИВОРЕЧИВОЕ ОПИСАНИЕ ИМЕНИ В  
COMMON-БЛОКЕ**

**SML64 НЕДОПУСТИМЫЙ ПЕРЕКЛЮЧАТЕЛЬ**

**SML65 НЕДОПУСТИМЫЙ ЭЛЕМЕНТ  
ПЕРЕКЛЮЧАТЕЛЬНОГО СПИСКА:**

⟨идентификатор⟩

**SML66 СОВПАДЕНИЕ ИМЕН КЛАССА И АТТРИБУТА  
В ЦЕПОЧКЕ ПРЕФИКСОВ ДЛЯ ⟨идентификатор⟩**

**SML67 НЕВЕРНЫЙ ЗАГОЛОВОК ПРОЦЕДУРЫ ИЛИ КЛАССА**

**SML68 НЕВЕРНОЕ УПОТРЕБЛЕНИЕ ОСНОВНОГО СИМВОЛА <идентификатор>**

**SML69 НЕ ОПИСАН ИДЕНТИФИКАТОР <идентификатор>**

Указанный идентификатор не описан ни в одном из блоков, текстуально охватывающих место его употребления. Выдав такое сообщение, компилятор считает, что этот идентификатор описан в текущем блоке как простая переменная типа **real**. Таким образом, фиксируется лишь первое употребление неопisanного идентификатора. Возможна наведенная диагностика, если для идентификатора подразумевалось другое описание.

**SML70 ПЕРЕПОЛНЕНА ТАБЛИЦА ДОСТУПНЫХ ИМЕН**

Необходимо уменьшить количество различных идентификаторов, описанных в блоках, текстуально охватывающих транслируемый участок. Имена, не используемые одновременно, нужно описать в параллельных блоках. Применение раздельной компиляции часто позволяет разгрузить таблицу имен.

**SML71 ПОВТОРНОЕ ОПИСАНИЕ ИДЕНТИФИКАТОРА <идентификатор>**

Один идентификатор дважды или более описан в том же блоке. Воспринимается лишь первое описание.

**З а м е ч а н и е.** Тексты сообщений, выдаваемые компилятором на БЭСМ-6, могут иметь незначительные отличия от приведенных в данном приложении.

## Приложение 5

### КОДИРОВКА ОСНОВНЫХ СИМВОЛОВ И ОПЕРАЦИЙ ДЛЯ ТРАНСЛЯТОРОВ НА БЭСМ-6 И ЕС ЭВМ

Основной символ	Кодировка	Основной символ	Кодировка
A — Z	A — Z	=	=
Б — Я	Б — Я	>	> или 'GT'
0 — 9	0 — 9	≥	'GE'
+	+	≠	'NE'
—	—	≡	'EQU'
*	*	⊃	'IMPL'
/	/	∨	'OR'
÷	'DI' или '**'	∧	'AND'
↑	** или 'POW'	⌋	'NOT'
<	< или 'LT'	, (запятая)	,
≤	'LE'	.	.



Основной символ	Кодировка	Основной символ	Кодировка
10	⊗, ◇ или E*	goto	'GOTO'
;	;	if	'IF'
: =	: =	in	'IN'
(	(	inner	'INNER'
)	)	inspect	'INSPECT'
[	[	integer	'INTEGER'
]	]	is	'IS'
' (апостроф)	'	label	'LABEL'
: —	% — или : — *	name	'NAME'
" (кавычки)	" или ? *	new	'NEW'
==	==	none	'NONE'
=/=	=/=	notext	'NOTEXT'
activate	'ACTIVATE'	otherwise	'OTHERWISE'
after	'AFTER'	prior	'PRIOR'
array	'ARRAY'	procedure	'PROCEDURE'
at	'AT'	qua	'QUA'
before	'BEFORE'	reactivate	'REACTIVATE'
begin	'BEGIN'	real	'REAL'
boolean	'BOOLEAN'	ref	'REF'
character	'CHARACTER'	step	'STEP'
class	'CLASS'	switch	'SWITCH'
code	'CODE'	text	'TEXT'
comment	'COMMENT'	then	'THEN'
delay	'DELAY'	this	'THIS'
do	'DO'	true	'TRUE'
external	'EXTERNAL'	until	'UNTIL'
else	'ELSE'	value	'VALUE'
end	'END'	virtual	'VIRTUAL'
false	'FALSE'	when	'WHEN'
for	'FOR'	while	'WHILE'

З а м е ч а н и е. Звездочкой (\*) помечены кодировки операций, допустимые только для ЕС ЭВМ.

## П р и л о ж е н и е 6

### ТАБЛИЦЫ РАНГОВ ЛИТЕР ДЛЯ БЭСМ-6 И ЕС ЭВМ

Т а б л и ц а (для БЭСМ-6)

	0	1	2	3	4	5	6	7	8	9
2		—	↑	10	≠	·	÷	,	⊃	≡
3	√	⌋		!	"	!	◇	%	∧	'
4	(	)	*	+	,	—	·	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>		!	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	!		—	—	Ю	А	Б	Ц
10	Д	Е	Ф	Г	Х	И	Й	К	Л	М
11	Н	О	П	Я	Р	С	Т	У	Ж	В
12	Ь	Ы	З	Ш	Э	Щ	Ч			

Т а б л и ц а (для ЕС ЭВМ)

	0	1	2	3	4	5	6	7	8	9
6										
7					[	.	<	(	+	!
8	&									
9		⊗	*	)	;	⌊	—	/		
10								,	%	—
11	>	?								
12			:	#	@	,	=	''		
13										
14										
15										
16										
17										
18					Ю		Б	Ц	Д	
19	Ф	Г		А	В	С	Д	Е	Ф	Г
20	Н	І		И	Й		Л			Ј
21	К	Л	М	Н	О	Р	Q	Р		
22	П	Я					S	T	U	V
23	W	X	Y	Z		У	Ж		Ь	Ы
24	0	1	2	3	4	5	6	7	8	9
25	з	ш	э	щ	ч					

## Приложение 7

### СООБЩЕНИЯ ДИАЛОГОВОГО ОТЛАДЧИКА

В приложении приводятся сообщения отладчика и действия пользователя в ответ на них.

#### 1. СИМВОЛЬНЫЙ ОТЛАДЧИК СИМУЛА-ЕС.

ВЕРСИЯ <номер>

ИМЯ ОТЛАЖИВАЕМОЙ ПРОГРАММЫ <имя>

Сообщение носит информационный характер. Оно выдается в начале работы отладчика и после каждой директивы НОВ. Имя программы, указанное в нем, становится текущим именем. После выдачи сообщения на экране дисплея появляется приглашение (==), которое означает, что можно приступить к отладке программы (программа готова к исполнению).

#### 2. КТ <контрольная точка> <имя программы>

Это сообщение означает, что после исполнения директивы ИДИ управление достигло одной из контрольных точек, ранее установленных с помощью директивы КТ. Кроме контрольной точки также указывается имя той программы, в которой она установлена.

### 3. КТ НЕ НАЙДЕНА

Сообщение сигнализирует об ошибке и появляется при попытке установления или отмены контрольной точки, если она указана в качестве операнда директивы, и не найдена в текущей программе. Причиной возникновения ошибки могут быть: а) отсутствие среди операторов программы указанной контрольной точки, б) выход вычисленного адреса контрольной точки за пределы поля программ [7] при ее задании с помощью относительного адреса, в) попытка установления контрольной точки по номеру оператора не в симула-программе (например, в фортран-программе, вызываемой из симула-программы).

Действия пользователя в ответ на это сообщение: а) проверить правильность задания контрольной точки при наборе директивы, б) проверить соответствие текущего имени программы, с именем программы, в которой устанавливается контрольная точка, в) установить контрольную точку с помощью относительного адреса.

### 4. СЧЕТНЫЙ РЕЖИМ

Это сообщение выдается при попытке установления или отмены контрольной точки по номеру оператора в программе, оттранслированной в счетном режиме. Чтобы продолжить работу, необходимо использовать второй способ задания контрольной точки (см. 4.6.2, директива КТ).

### 5. ЛИШНЯЯ КТ

Данное сообщение выдается при попытке установления контрольной точки, а в таблице контрольных точек уже имеется 10 точек. Для установления новой контрольной точки необходимо исключить хотя бы одну контрольную точку из имеющихся в таблице, с помощью директивы ОТМ.

### 6. АДРЕС НЕ КРАТЕН 2

Сообщение выдается, если относительный адрес оператора, с помощью которого задается контрольная точка, не кратен 2.

### 7. АВОСТ ОС <код прерывания> ПО АДРЕСУ <адрес>.

Выдача этого сообщения происходит во время выполнения программы при обнаружении ошибок, вызывающих программные прерывания. Каждое программное прерывание имеет шестнадцатеричный код — код прерывания. Прерывания с кодами 8 (прерывание при выполнении операции с фиксированной точкой), А (прерывание при выполнении операции с десятичной арифметикой), D (исчезновение порядка при выполнении операции с плавающей запятой), Е (потеря значимости при выполнении операции с плавающей запятой), — замаскированы, т. е. при их возникновении выполнение программы продолжается.

Кроме кода прерывания в сообщении указывается абсолютный адрес памяти, где обнаружена ошибка. После выдачи та-

кого сообщения дальнейшее продолжение отладки программ невозможно.

#### 8. ОТСУТСТВУЕТ ИМЯ <имя программы>

Сообщение выдается при попытке изменения текущего имени, если в качестве операнда у директивы ИМЯ указано имя программы, отсутствующее в листе загрузки. Текущее имя не изменяется. Для изменения текущего имени достаточно еще раз повторить директиву с правильным операндом.

#### 9. НЕТ ТАКОЙ ДИРЕКТИВЫ

Выполнение не имеющейся в отладчике директивы, приводит к выдаче данного сообщения. Директива игнорируется и выдается новое приглашение к работе.

#### 10. ИСПОЛЬЗОВАНИЕ СИМВОЛА ' ' НЕДОПУСТИМО

Сообщение появляется при обнаружении синтаксической ошибки в записи директивы или ее операндов. В кавычках указывается недопустимый в директиве символ. Символ «конец текста» (нераспечатываемый) может быть заменен на точку с запятой. В этом случае следует исправить ошибку и продолжить работу.

#### 11. НЕВЕРНЫЕ ЗНАЧЕНИЯ ПАРАМЕТРОВ

Сообщение выдается при неверном задании параметров в операторах отладки trace и evq, а сами операторы игнорируются. Следует исправить ошибку и продолжить работу.

#### 12. КОНЕЦ ПРОГРАММЫ

Выдача сообщения связана с выходом управления через завершающий символ end главной программы. После выдачи сообщения освобождается оперативная память, занятая ранее под отлаживаемый комплекс программ. Для возобновления отладки необходимо набрать директиву НОВ.

#### 13. В ПАМЯТИ НЕТ ПРОГРАММЫ

Это сообщение выдается при попытке использования директив ВЫВ, УСТ, ИДИ, КТ, если отлаживаемый комплекс программ не находится в оперативной памяти машины. Такая ситуация возможна при условии выдачи предыдущего сообщения (12).

#### 14. РАНО

Использование директив ПД, EVQ, ВЫВ, УСТ, ПВ за собой выдачу этого сообщения, если главная программа еще не начала исполняться. Использование перечисленных директив возможно только после останова в контрольной точке, т. е. после получения на экране сообщения под номером 2.

#### 15. КОНЕЦ СЕАНСА

Данное информационное сообщение выдается после нормального завершения работы отладчика.

## 16. ТЕКСТ УКОРОЧЕН НА: <целое>

Это сообщение появляется при присваивании тексту значения, превышающего его длину. Текст укорачивается и в конец его заносится знак вопроса (?). Сообщение является предупреждающим, выполнение отлаживаемой программы продолжается.

17. НЕ СОВПАДАЕТ РАЗМЕРНОСТЬ МАССИВА <идентификатор> и ИНДЕКС ВЫХОДИТ ЗА ГРАНИЦЫ МАССИВА <идентификатор>.

Эти сообщения появляются при ошибочном обращении к индексируемой переменной. При этом интерпретация директивы не заканчивается: если сообщение появилось при выполнении директивы ВЫВ или касается переменной в правой части оператора присваивания в директиве УСТ, то значение переменной считается равным начальному (0, false, none или notext в зависимости от типа переменной), а если оно указывает на ошибку в задании левой части оператора присваивания, то значение левой части не изменяется. В указанных случаях следует внимательно следить за значениями переменных.

## 18. ЗНАЧЕНИЕ ЛЕВОЙ ЧАСТИ-NOTEXT

Сообщение выдается при попытке присвоить текстовой переменной, значение которой notext, текстовое значение или текстовую ссылку.

## 19. ЗАПИСЬ ЧИСЛА ПРЕВЫШАЕТ МАКСИМАЛЬНОЕ ДОПУСТИМОЕ ЗНАЧЕНИЕ

Появляется, когда в тексте директивы встречается число, которое не может быть переведено во внутреннее представление, например: оператор присваивания X:=9999999999999999, невыполним, если X — целая переменная, однако, выполним, если X — вещественная переменная.

## 20. НЕСООТВЕТСТВИЕ ТИПА ОПЕРАНДА

Выдается при попытке присвоить переменной значение неподходящего типа. В этом случае надо исправить ошибку и повторить действие.

## 21. ЗНАЧЕНИЕ ОБЪЕКТНОГО ВЫРАЖЕНИЯ-NONE

Сообщение выдается, если в дистанционном идентификаторе значение промежуточной ссылочной переменной равно none. В этом случае надо исправить ошибку и повторить действия.

## 22. ИМЯ <идентификатор> НЕ НАЙДЕНО

Сообщение выдается, если соответствующая переменная не описана в данной точке программы или недоступна через цепочку ссылок. В этом случае надо исправить ошибку или остановиться в такой точке программы, где переменная доступна.

# Приложение 8. ПРИМЕР ДИАЛОГА

С И М У Л А - 67. ВЕРСИЯ 3.1 ИПМ АН СССР - МИФИ. 18.03.83.

0.0 ИМЯ SERVICE, РЕЖ=ОТЛ;

```

1.0 'BEGIN' 'REAL' TM; TOX, CBOX, A, ЗБ, M, S; 'INTEGER' K, KA, KOA, HEJO, U1, U2; 'INTEGER' KЗМ;
1.1 TM:=INREAL; M:=INREAL; S:=INREAL; A:=INREAL; K=ININT; OUTTEXT('МОДЕЛИРОВАНИЕ СТАНЦИИ ТЕХНИЧЕСКОГО '); OUTTEXT('ОБ
1.7 СЛУЖИВАНИЯ АВТОМОБИЛЕЙ'); OUTIMAGE; OUTTEXT('ВРЕМЯ РАБОТЫ ИНТЕНСИВНОСТЬ ПОТОКА '); OUTTEXT(' ПАРАМЕТРЫ ОБСЛУЖИВ
1.10 АНИЯ '); OUTIMAGE; OUTFIX(TM, 2, 10); OUTFIX(A, 2, 15); OUTTEXT(' '); OUTTEXT('M-'); OUTFIX(M, 2, 8); OUTTEXT
1.17 ('S-'); OUTFIX(S, 2, 8); OUTTEXT('K-'); OUTINT(K, 5); OUTIMAGE; TRACE(0, 10); ПРОГОН; SIMULATION
2.0 'BEGIN' 'REF'(БРИГАДА)БРИГ; 'REF'(HEAD)СТОЯНКА;
3.0 PROCESS'CLASS'А В Т О М О Б И Л Ь,
4.0 'BEGIN' 'REAL' Н О Ж,
4.1 КА := КА + 1; 'IF' КЗМ = К 'THEN' 'GOTO' КОНЕЦ; Н О Ж := TIME; 'THIS' АВТОМОБИЛЬ INTO(СТОЯНКА); КЗМ := КЗ
4.5 М + 1; 'ACTIVATE' БРИГ 'AFTER' CURRENT; КОНЕЦ; 'ACTIVATE' 'NEW' АВТОМОБИЛЬ 'DELAY' NEGEXP( A, U1);
4.9 'END'А В Т О М О Б И Л Ь;
5.0 PROCESS'CLASS'Б Р И Г А Д А,
6.0 'BEGIN' 'REAL' НРАБ, ТРАБ, ЖДАЛ; 'REF'(АВТОМОБИЛЬ)КЛИЕНТ,
6.1 РАБОТА 'IF' СТОЯНКА EMPTY 'THEN' PASSIVATE; НРАБ := TIME; КЛИЕНТХ-СТОЯНКА FIRST; КЛИЕНТ OUT. ЖДАЛ = TIME
6.5 - КЛИЕНТ. Н О Ж, 'IF' ЖДАЛ = 0 'THEN' НЕЖО = НЕЖО + 1
6.6 'ELSE' ТОЖ := ТОЖ + ЖДАЛ; КЗМ := КЗМ - 1; HOLD(NORMAL M, S, U2); ТРАБ = ТРАБ + (TIME - НРАБ); КОА = КОА + 1
6.10 Г О Т О РАБОТА,
6.13 'END'Б Р И Г А Д А;
2.1 U1 = 1; U2 = 3; КЗМ = КА = КОА = НЕЖО = CBOX = 0; СТОЯНКА = 'NEW' HEAD; БРИГ = 'NEW' БРИГАДА; 'ACTIVATE' 'NEW' АВТОМОБИЛЬ;
2.7 Н О Л Д (Т М), ЗБ = БРИГ ТРАБ / ТМ,
2.10 'END' БЛОКА SIMULATION;
1.24 'IF' КОА / КА < 0.8 'THEN'
7.0 'BEGIN' A := 0.9 * A; 'GOTO' ПРОГОН
7.3 'END',
1.25 OUTTEXT('ИСКАМОЯ ИНТЕНСИВНОСТЬ '); OUTFIX(A, 2, 10); OUTIMAGE; 'IF' НЕЖО < КОА 'THEN' CBOX = ТОЖ / (КОА - НЕЖО); OUTTEX
1.29 T(' РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ '); OUTIMAGE; OUTTEXT('ПРИШЛО АВТОМОБИЛЕЙ - '); OUTINT(KA, 6); OUTTEXT(' ИЗ НИХ ОБ
1.33 СЛУЖЕНО - '); OUTINT(KOA, 6); OUTTEXT(' НЕ ЖДАЛИ - '); OUTINT(HEJO, 6); OUTIMAGE; OUTTEXT(' СРЕДНЕЕ ВРЕМЯ ОЖИДАНИЯ
1.38 - '); OUTFIX(CBOX, 2, 10); OUTIMAGE; OUTTEXT('ЗАГРУЗКА БРИГАДЫ - '); OUTFIX(ЗБ, 2, 5);
1.44 'END'
0.3 'EOP'

```

## ОТНОСИТЕЛЬНЫЕ АДРЕСА ОПЕРАТОРОВ

1 1	00001E	1 2	00005B	1 3	000062	1 4	00006C	1 5	000076
1 6	000080	1 7	00008A	1 8	000094	1 9	00009A	1 10	0000A4
1 11	0000AE	1 12	0000B4	1 13	0000D2	1 14	0000F0	1 15	0000FA
1 16	000104	1 17	000122	1 18	00012C	1 19	00014A	1 20	000154
1 21	00016A	1 22	000170	1 23	000186	1 24	00019C	7 1	0001DB
7 2	0001E4	1 25	0001F0	1 26	0001FA	1 27	000218	1 28	00021E
1 29	000268	1 30	000272	1 31	000278	1 32	000282	1 33	000298
1 34	0002A2	1 35	0002B8	1 36	0002C2	1 37	0002D8	1 38	0002DE
1 39	0002EB	1 40	000306	1 41	00030C	1 42	000316	1 43	000334
2 1	000402	2 2	00040A	2 3	000412	2 4	000436	2 5	000460
2 6	00048C	2 7	0004B4	2 8	0004CE	2 9	0004E4	4 1	00053A
4 2	000546	4 3	00057A	4 4	000586	4 5	0005AC	4 6	0005B8
4 7	0005DC	4 8	000622	6 1	00067E	6 2	0006AE	6 3	0006BA
6 4	0006E2	6 5	0006F4	6 6	000716	6 7	000766	6 8	000772
6 9	0007A6	6 10	0007BA	6 11	0007C6	6 12	0007D2		

ТРАНСЛЯЦИЯ ЗАКОНЧЕНА. 6681

ЧИСЛО ОШИБОК=0

ДЛИНА ОБЪЕКТА МОДУЛЯ (В БАЙТАХ)=3405

ПАМЯТЬ: 5120

СИМВОЛЬНЫЙ ОТСЛАДЧИК СИМУЛА-ЕС. ВЕРСИЯ 2.0

ИМЯ ОТЛАЖИВАЕМОЙ ПРОГРАММЫ SERVICE

= DC 12B

= KT 2.1

= ИЛИ

KT 2.1 SERVICE

= TRACE(0,5)

= KT 6.11

= TAB

6.11 SERVICE

= ИЛИ

KT 6.11 SERVICE

= KT 1.24

= ИЛИ

KT 1.24 SERVICE

= ВВБ А, КА, КОА

5.0000000E+00

710

13

= KT 7.2

= ИЛИ

KT 7.2 SERVICE

= ВВБ А

4.4999990E+00

= УСТ А:=0,3

= KT 1.24

```

"      ВЫБ КА. КОД
      710
      13
"      КТ 1Е4
"      КТ 4 1
"      ТАБ
1.24 SERVICE 0001Е4 SERVICE 4 1
"      ИДИ
КТ 0001Е4 SERVICE
"      ВЫБ А
      3 0000000Е-01
"      ИДИ
КТ 4 1. SERVICE
"      ВЫБ БРИГ
000ВA1В4 ОБЪЕКТ КЛАССА БРИГАДА 721
"      ИДИ
КТ 1 24 SERVICE
"      КТ 1 43
"      ИДИ
КТ 1 43 SERVICE
"      ПБ

```

РАЗМЕР ПАМЯТИ, ВЫДЕЛЕННОЙ ПОД ПО (В БАЙТАХ) :

100000

АДРЕС НАЧАЛА ПАМЯТИ : 07A960

АДРЕС КОНЦА ПАМЯТИ . 092FFC

МОДЕЛИРОВАНИЕ СТАНЦИИ ТЕХНИЧЕСКОГО ОБСЛУЖИВАНИЯ АВТОМОБИЛЕЙ

ВРЕМЯ РАБОТЫ ИНТЕНСИВНОСТЬ ПОТОКА ПАРАМЕТРЫ ОБСЛУЖИВАНИЯ

144.00 5.00 M= 10.50 S= 1.35 K= 3

\*\*\* УПРАВЛЕНИЕ В ПРОГРАММЕ SERVICE \*\*\*

```

* 1 23 * ГЕНЕРАЦИЯ : КПС SIMULAT БРИГ=NONE СТОЯНКА=NONE SGS=NONE REFFП=NONE SVCS=NONE PREDS=NONE
      EV=NONE T1=FALSE
* 2 4 * ГЕНЕРАЦИЯ ОБЪЕКТ HEAD 4 SUCC=NONE PREDD=NONE
* 2 5 * ГЕНЕРАЦИЯ ОБЪЕКТ БРИГАДА 5 НРАБ= 0.0000000Е+00 ТРАБ= 0.0000000Е+00 ЖДАЛ= 0.0000000Е+00 КЛИЕНТ=NONE
      EVENT=NONE TERM1=FALSE SUCC=NONE PREDD=NONE
* 2 6 * ГЕНЕРАЦИЯ: ОБЪЕКТ АВТОМОБ 6 НОЖ= 0.0000000Е+00 EVENT=NONE TERM1=FALSE SUCC=NONE PREDD=NONE

* 4 2 * УСЛОВИЕ FALSE
* 4.7 * TIME= 0.0000000Е+00 ПРОЦЕСС АВТОМОБ 6 НОЖ= 0.0000000Е+00 EVENT=EVENTNO 0 TERM1=FALSE SUCC=HEAD 4
      PREDD=HEAD 4
** УПР. СПИСОК
0.0000000Е+00 * (SERVICE) ПРОЦЕСС АВТОМОБ 6 НОЖ= 0.0000000Е+00 EVENT=EVENTNO 0 TERM1=FALSE
      SUCC=HEAD 4 PREDD=HEAD 4
0.0000000Е+00 * 6.1 * ПРОЦЕСС БРИГАДА 5 НРАБ= 0.0000000Е+00 ТРАБ= 0.0000000Е+00 ЖДАЛ= 0.0000000Е+00
      КЛИЕНТ=NONE EVENT=EVENTNO 0 TERM1=FALSE SUCC=NONE PREDD=NONE
0.0000000Е+00 * 2.6 * ПРОЦЕСС SIMULAT БРИГ=БРИГАДА 5 СТОЯНКА=HEAD 4 SGS=HEAD 2 REFFП=SIMULAT SVCS=EVENTNO 3
"      ИДИ

```



## КОДЕС ПРОГРАММЫ

= НОВ

1

## OC EC LOADER

OPTIONS USED - PRINT, MAP, LET, CALL, NORES, NOTERM, SIZE=150000, NAME==GO

NAME	TYPE	ADDR	NAME	TYPE	ADDR	NAME	TYPE	ADDR	NAME	TYPE	ADDR	NAME	TYPE	ADDR
0 SERVICE	SD	69620	SFIN	* SD	6A370	SYSINC	* SD	6A520	FILE	* LR	6A6EC	INFILE	* LR	6A738
OUTFILE	* LR	6A784	PRINTFI	* LR	6A82C	SSIMSET	* SD	6AA70	SIMSE	* LR	6AB0C	LINKAGE	* LR	6AB70
HEAD	* LR	6ABEC	LINK	* LR	6AC38	OSIML	* SD	6ACF0	SIMULAT	* LR	6AE60	ACCUM	* LR	6AF48
EVENTNO	* LR	6AFB8	PROCESS	* LR	6B03C	SBEQP	* SD	6B1D0	TRAW	* LR	6B578	SVXB	* SD	6B5B0
SDVXB0	* SD	6B6C8	FTNREAL	* SD	6B7D0	INREAL	* LR	6B8C4	FININT	* SD	6B8E0	ININT	* LR	6B9D4
FOUTTEX	* SD	6B9E8	OUTTEXT	* LR	6BA6C	FOUTIMA	* SD	6BAE8	OUTIMAG	* LR	6BC48	FOUTFIX	* SD	6BC70
OUTFIX	* LR	6BCD4	FOUTINT	* SD	6BD68	OUTINT	* LR	6BDCC	TRACE	* SD	6BE58	SMAOP	* SD	6BF20
SGENKP	* SD	6BFF8	SKPTS	* SD	6C0E8	SIVR	* SD	6C1D8	SIMIF	* SD	6C208	SGOOT	* SD	6C310
SDOUTB	* SD	6C420	SYPZ	* SD	6C5A0	SRVI	* SD	6C600	SGENOB	* SD	6C648	SVXK	* SD	6C750
ACTIV	* SD	6C880	REACT	* LR	6C8EE	TPAKT	* SD	6C980	HOLDOT	* SD	6CAE0	SPNONE	* SD	6CC00
SGOKN	* SD	6CC90	TIME	* SD	6CCB0	THIS	* SD	6CCDB	PRECEDE	* SD	6CDC8	INTO	* LR	6CDC6
CURRENT	* SD	6CEA0	NEGEXP	* SD	6CED0	SYPK	* SD	6CF68	STPKOH	* SD	6CFB0	EMPTY	* SD	6D090
PASSOT	* SD	6D0A8	SUC	* SD	6D198	FIRST	* LR	6D198	S67PCP	* SD	6D238	OUT	* SD	6D2E8
NORMAL	* SD	6D320	SPRTEXT	* SD	6D3C8	SPRINS	* LR	6D438	BUF	* LR	6D509	S10T	* SD	6D5A8
CYSIN	* SD	6D608	ASYSIN	* LR	6D694	AINFL	* LR	6D750	SBIXK	* SD	6D7B0	SOUTP	* SD	6D8C0
SELPR	* SD	6D8D0	SIDN	* SD	6D8D8	SDETIN	* LR	6D9F8	PASSIVA	* SD	6DA20	DETACH	* SD	6DAF8
CYSOUT	* SD	6DC10	ASYSOUT	* LR	6DC9C	SYSPRIN	* LR	6DD94	GETINT	* SD	6DDF8	S16T	* SD	6DF20
SABOC	* SD	6DF78	SOBLCOX	* LR	6DFB8	SDAYPAM	* SD	6E008	SNUMOP	* SD	6E1C0	FREOBBU	* SD	6E338
SLASTIT	* SD	6E470	SUB	* SD	6E588	GETREAL	* SD	6E6E0	S60TPA	* SD	6E6D8	FIELD	* SD	6E9B0
S6PRTZ	* SD	6EAC0	SPRFILE	* SD	6EBF8	PUTFIX	* SD	6ECA0	PUTINT	* SD	6EE46	SAB	* SD	6EF48
PIOD	* SD	6FBA8	SDDISP	* SD	6FF38	SGOTO	* SD	6FFA8	SDOSTR	* SD	70150	SDESTR	* LR	70166
SOSOB	* SD	70268	STFOFA	* SD	702C0	SPDCBN	* LR	7044E	SDACTH	* SD	70490	SDRCH	* SD	70558
SDACT	* SD	705D0	SDRCT	* SD	70690	SAFTER	* SD	706E8	SBEFORE	* LR	707A0	SAT	* SD	7082Q
SDelay	* LR	7089E	SATP	* SD	708D0	SDelayP	* LR	70950	CONVER	* SD	70980	EV3	* SD	70A80
HOLD	* SD	70D88	IN	* SD	70E40	SNEGEX	* LR	70E88	SNORMA	* SD	70FF8	FINIMAG	* SD	71238
SEOF	* LR	712C2	INIMAG	* LR	7136C	RESUME	* SD	71388	CTEK	* SD	71480	SELEX	* SD	71610
SMAOPP	* SD	71710	SVXP	* SD	717D0	SBKBIK	* SD	71888	SINPUT	* SD	718C6	SOBPAB	* SD	718A8
SCLEAR	* SD	720F0	ORDK	* LR	72324	S6DAPA	* SD	72378	S6TZT	* SD	72618	SPRIZ	* LR	727AC
SPRINT	* SD	727C8	SIMIND	* SD	72BE0	SDUMP	* SD	72D80	SCORDI	* SD	727F0	SGOOT	* SD	73820
SZPYB	* SD	73878	PSRAND	* SD	738C8	IHCLOG	* SD	73918	ALOG10	* LR	73910	ALOG	* LR	73930
IHCSSGRT	* SD	73A80	SQRT	* LR	73A80	REG	* SD	73B90	SOBLC	* SD	73E58	SOBLCB	* SD	74028
SBORMYS	* SD	742D0	SIMSYM	* LR	74750	NIKEM	* SD	74760	PUTREAL	* SD	747D0	BLANKCOM	CM	74908

IEW1161

0 TOTAL LENGTH 836C

ENTRY ADDRESS 69620

-IEW1161 WARNING - NO ENTRY POINT RECEIVED

PREDS=NONE EV=NONE T1=FALSE

\* 4 7 \* ПЕРЕПИСЬ ОБЪЕКТ АВТОМОБ 7 НОМ= 0 0000000E+00 EVENT=NONE TERM1=FALSE SUCC=NONE PREDD=NONE

```

* 4 8 * TIME= 0.0000000E+00 ПРОЦЕСС АВТОМОБ 6 НОХ= 0.0000000E+00 EVENT=EVENTNO 0 TERM1=FALSE .SUCC=HEAD 4
      PREDD=HEAD 4
      ** УПР СПИСОК
0 0000000E+00 * (SERVICE) ПРОЦЕСС АВТОМОБ 6 НОХ= 0.0000000E+00 EVENT=EVENTNO 0 .TERM1=FALSE
      SUCC=HEAD 4 PREDD=HEAD 4
0 0000000E+00 * 6 1 * ПРОЦЕСС БРИГАДА 5 НРАБ= 0.0000000E+00 ТРАБ= 0.0000000E+00 ХДАЛ= 0.0000000E+00
      КЛИЕНТ=NONE EVENT=EVENTNO 0 TERM1=FALSE SUCC=NONE PREDD=NONE
0 0000000E+00 * 2 6 * ПРОЦЕСС SIMULAT БРИГ=БРИГАДА 5 СТОЯНКА=HEAD 4 SGS=HEAD 2 РЕФГП=SIMULAT SVCS=EVENTNO 3
      PRED=NONE EV=NONE T1=FALSE
3 9774700E-01 * 4.1 * ПРОЦЕСС АВТОМОБ 7 НОХ= 0.0000000E+00 EVENT=EVENTNO 0 TERM1=FALSE .SUCC=NONE
      PREDD=NONE
* 4 8 * ЗАВЕРШЕНИЕ ОБ'ЕКТ АВТОМОБ 6 НОХ= 0.0000000E+00 EVENT=EVENTNO 0 TERM1=FALSE SUCC=HEAD 4 PREDD=HEAD 4

* 6 1 * УСЛОВИЕ FALSE
* 6 6 * УСЛОВИЕ TRUE
* 6 8 * HOLD НА 1 0606260E+01 ПРОЦЕСС БРИГАДА 5 НРАБ= 0 0000000E+00 ТРАБ= 0 0000000E+00 ХДАЛ= 0 0000000E+00
      КЛИЕНТ=АВТОМОБ 6 EVENT=EVENTNO 0 TERM1=FALSE SUCC=NONE PREDD=NONE
* 2.7 * TIME= 0 0000000E+00 ПРОЦЕСС SIMULAT БРИГ=БРИГАДА 5 СТОЯНКА=HEAD 4 SGS=HEAD 2 РЕФГП=SIMULAT
      SVCS=EVENTNO 3 PRED=NONE EV=NONE T1=FALSE
      ** УПР. СПИСОК
0 0000000E+00 * (SERVICE) ПРОЦЕСС SIMULAT БРИГ=БРИГАДА 5 СТОЯНКА=HEAD 4 SGS=HEAD 2 РЕФГП=SIMULAT
      SVCS=EVENTNO 3 PRED=NONE EV=NONE T1=FALSE
3 9774700E-01 * 4.1 * ПРОЦЕСС АВТОМОБ 7 НОХ= 0.0000000E+00 EVENT=EVENTNO 0 TERM1=FALSE SUCC=NONE
      PREDD=NONE
1 0606260E+01 * 6 8 * ПРОЦЕСС БРИГАДА 5 НРАБ= 0 0000000E+00 ТРАБ= 0.0000000E+00 ХДАЛ= 0 0000000E+00
      КЛИЕНТ=АВТОМОБ 6 EVENT=EVENTNO 0 TERM1=FALSE SUCC=NONE PREDD=NONE
* 2 7 * HOLD НА 1 4400000E+02 ПРОЦЕСС SIMULAT БРИГ=БРИГАДА 5 СТОЯНКА=HEAD 4 SGS=HEAD 2 РЕФГП=SIMULAT
      SVCS=EVENTNO 3 PRED=NONE EV=NONE T1=FALSE
* 4 2 * УСЛОВИЕ FALSE
* 4 7 * ГЕНЕРАЦИЯ ОБ'ЕКТ АВТОМОБ 8 НОХ= 0.0000000E+00 EVENT=NONE TERM1=FALSE SUCC=NONE PREDD=NONE

* 4 8 * ЗАВЕРШЕНИЕ ОБ'ЕКТ АВТОМОБ 7 НОХ= 3.9774700E-01 EVENT=EVENTNO 0 TERM1=FALSE SUCC=HEAD 4 PREDD=HEAD 4

* 4 2 * УСЛОВИЕ FALSE
* 4 8 * ЗАВЕРШЕНИЕ ОБ'ЕКТ АВТОМОБ 8 НОХ= 4.4706890E-01 EVENT=EVENTNO 0 TERM1=FALSE SUCC=HEAD 4 PREDD=АВТОМОБ 7

* 4 8 * ЗАВЕРШЕНИЕ ОБ'ЕКТ АВТОМОБ 9 НОХ= 8 4514270E-01 EVENT=EVENTNO 0 TERM1=FALSE SUCC=HEAD 4 PREDD=АВТОМОБ 8

* 4.8 * ЗАВЕРШЕНИЕ ОБ'ЕКТ АВТОМОБ 10 НОХ= 0.0000000E+00 EVENT=EVENTNO 0 TERM1=FALSE SUCC=NONE PREDD=NONE

```

```

ИСКАЯ ИНТЕНСИВНОСТЬ = 0.12
РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ:
ПРИВЫЛО АВТОМОБИЛЕЙ - 13, ИЗ НИХ ОБСЛУЖЕНО - 12, НЕ ХДАЛ - 4
СРЕДНЕЕ ВРЕМЯ ОЖИДАНИЯ - 325.01
ЗАГРУЗКА БРИГАДЫ - 0.866

```

КОНЕЦ РАБОТЫ ПРОГРАММЫ  
 МАХ РАЗМЕР ПАМЯТИ, ЗАНЯТОЙ ПОД ПО (В БАЙТАХ)

92628

ИМЯ ОТЛАЖИВАЕМОЙ ПРОГРАММЫ SERVICE

= КТ 2.1

= ИДИ

КТ 2 1 SERVICE

= TRACE(0.0)

= ВЫВ К, А

3

5.0000000E+00

= УСТ К:=6.А =0 4

= ВЫВ К, А

6

4.0000000E-01

= КТ 1.43

= ИДИ

КТ 1 43 SERVICE

= ПБ

РАЗМЕР ПАМЯТИ, ВЫДЕЛЕННОЙ ПОД ПО (В БАЙТАХ):

100000

АДРЕС НАЧАЛА ПАМЯТИ : 07A960

АДРЕС КОНЦА ПАМЯТИ : 092FFC

МОДЕЛИРОВАНИЕ СТАНЦИИ ТЕХНИЧЕСКОГО ОБСЛУЖИВАНИЯ АВТОМОБИЛЕЙ

ВРЕМЯ РАБОТЫ ИНТЕНСИВНОСТЬ ПОТОКА ПАРАМЕТРЫ ОБСЛУЖИВАНИЯ

144.00 5.00 M= 10.50 S= 1 35 K= 3

\*\*\* УПРАВЛЕНИЕ В ПРОГРАММЕ SERVICE \*\*\*

\* 1 23 \* ГЕНЕРАЦИЯ : КПС SIMULAT БРИГ=NONE СТОЯНКА=NONE SGS=NONE REFTG=NONE SVCS=NONE PRED=NONE

EV=NONE T1=FALSE

ИСКАЯМАЯ ИНТЕНСИВНОСТЬ = 0.13

РЕЗУЛЬТАТЫ МОДЕЛИРОВАНИЯ:

ПРИБЫЛО АВТОМОБИЛЕЙ - 14, ИЗ НИХ ОБСЛУЖЕНО - 13, НЕ ЖДАЛИ - 3

СРЕДНЕЕ ВРЕМЯ ОЖИДАНИЯ - 492.80

= КОИ

1 OC EC LOADER

OPTIONS USED - PRINT, MAP, LET, CALL, NORES, NOTERM, SIZE=150000, NAME=\*\*GO

	NAME	TYPE	ADDR	NAME	TYPE	ADDR	NAME	TYPE	ADDR	NAME	TYPE	ADDR	NAME	TYPE	ADDR
0	SERVICE	SD	69620	SFIN	* SD	6A370	SYSINO	* SD	6A5E0	FILE	* LR	6A6EC	INFILE	* LR	6A738
	OUTFILE	* LR	6A784	PRINTFI	* LR	6A82C	SSIMSET	* SD	6AA78	SIMSET	* LR	6AB0C	LINKAGE	* LR	6AB70
	HEAD	* LR	6ABEC	LINK	* LR	6AC38	SSIMLTN	* SD	6ACF0	SIMULAT	* LR	6AE60	ACCUM	* LR	6AF48
	EVENTNO	* LR	6AF88	PROCESS	* LR	6B03C	SBEQP	* SD	6B1D0	TRAW	* LR	6B578	SVXB	* SD	6B5B0
	SDVXB0	* SD	6B6C8	FINREAL	* SD	6B7D0	INREAL	* LR	6B8C4	FININT	* SD	6B8E0	ININT	* LR	6B9D4
	FOUTTEX	* SD	6B9E8	OUTTEXT	* LR	6BA6C	FOUTIMA	* SD	6BAEB	OUTIMAG	* LR	6BC48	FOUTFIX	* SD	6BC70
	OUTFIX	* LR	6BCD4	FOUTINT	* SD	6BD68	OUTINT	* LR	6BDCC	TRACE	* SD	6BE58	SMAOP	* SD	6BF20
	SGENKF	* SD	6BFF8	SKPS	* SD	6C0E8	SIVR	* SD	6C1D8	SIMIF	* SD	6C208	SQOOT	* SD	6C310

SDOUT3	*	SD	6C420	SYPZ	*	SD	6C5A0	SRVI	*	SD	6C600	SGENOB	*	SD	6C648	SVXK.	*	SD	6C750
ACTIV	*	SD	6C880	REACT	*	LR	6C8EE	TPAKT	*	SD	6C980	HOLDOT	*	SD	6CAE0	SPNONE	*	SD	6CC00
SGOKN	*	SD	6CC90	TIME	*	SD	6CCB0	THIS	*	SD	6CCDB	PRECED	*	SD	6CDC8	INTO	*	LR	6CDC8
CURRENT	*	SD	6CEA0	NEGEXP	*	SD	6CED0	SYPK	*	SD	6CF68	STPKOH	*	SD	6CFE0	EMPTY	*	SD	6D090
PASSOT	*	SD	6D0A8	SUC	*	SD	6D198	FIRST	*	LR	6D198	S67PCP	*	SD	6D238	DUT	*	SD	6D2E8
NORMAL	*	SD	6D320	SPRTEXT	*	SD	6D3CB	SPRINS	*	SD	6D438	BUF	*	LR	6D509	S10T	*	SD	6D5A8
CYSIN	*	SD	6D608	ASYSIN	*	LR	6D694	AINFL	*	LR	6D750	SBIXK	*	SD	6D780	SOUTP	*	SD	6D8C0
SGLPR	*	SD	6D8D0	SIDN	*	SD	6D8D8	SDETIN	*	SD	6D9F8	PASSIVA	*	SD	6DA20	DETACH	*	SD	6DAF8
CYSOUT	*	SD	6DC10	ASYSOUT	*	LR	6DC9C	SYSPRIN	*	LR	6DD94	GETINT	*	SD	6DDF8	S16T	*	SD	6DF20
SABOC	*	SD	6DF78	SOBLCOX	*	LR	6DF88	SDAYPAM	*	SD	6E008	SNUMOP	*	SD	6E1C0	PREOBBU	*	SD	6E338
SLASTIT	*	SD	6E470	SUB	*	SD	6E588	GETREAL	*	SD	6E6E0	S60TPA	*	SD	6E8D8	FIELD	*	SD	6E980
S6PRTZ	*	SD	6EAC0	SPRFILE	*	SD	6EBF8	PUTFIX	*	SD	6ECA0	PUTINT	*	SD	6EE48	SAB	*	SD	6EF48
PID0	*	SD	6F8A8	SDDISP	*	SD	6FF38	SGOTO	*	SD	6FFA8	SDOSTR	*	SD	70150	SDESTR	*	LR	70166
S0S0B	*	SD	70268	STFOFA	*	SD	702C0	SPDCBN	*	LR	7044E	SDACTH	*	SD	70490	SDRCTH	*	SD	70558
SDACT	*	SD	705D0	SDRCT	*	SD	70690	SAFTER	*	SD	706E8	SBEFORE	*	LR	707A0	SAT	*	SD	70820
SDELAY	*	LR	7089E	SATP	*	SD	708D0	SDELAYP	*	LR	70950	CONVER	*	SD	70980	EVG	*	SD	70A80
HOLD	*	SD	70DB8	IN	*	SD	70E40	SNEGEX	*	SD	70E88	SNORMA	*	SD	70FF8	FINIMAG	*	SD	71238
SEOF	*	LR	712C2	INIMAGE	*	LR	7136C	RESUME	*	SD	71388	CTEK	*	SD	71480	SBLEX	*	SD	71610
SMAOPP	*	SD	71710	SVXP	*	SD	717D0	SBKBIK	*	SD	71888	SINPUT	*	SD	718C8	SOBPAB	*	SD	718A8
SCLEAR	*	SD	720F0	ORDK	*	LR	72324	S6DAPA	*	SD	72378	S6ZTZ	*	SD	72618	SPRIZ	*	LR	727AC
SPRINT	*	SD	727C8	SIMIND	*	SD	728E0	SDUMP	*	SD	72DB0	SCORDI	*	SD	737F0	SCOCT	*	SD	73820
SZPYB	*	SD	73878	PSRAND	*	SD	738C8	IHCSSLOG	*	SD	73918	ALOG10	*	LR	73918	ALOG	*	LR	73930
IHCSSGRT	*	SD	73A80	S8RT	*	LR	73A80	REQ	*	SD	73B90	SOBLC	*	SD	73E58	SOBLCB	*	SD	74028
SBORMYS	*	SD	742D0	SIMSYM	*	LR	74750	NIKEM	*	SD	74760	PUTREAL	*	SD	747D0	\$BLANKCOM	CM		74908

IEW1161

0 TOTAL LENGTH B36C

ENTRY ADDRESS 69620

-IEW1161 WARNING - NO ENTRY POINT RECEIVED

KOHEU CEAHCA

## Приложение 9. ПРИМЕРЫ СИМУЛА-ПРОГРАММ 9.1. Программа расстановки ферзей

СИМУЛА - 67. ВЕРСИЯ 3.1 ИГМ АН СССР - МИФИ. 18.03.83.

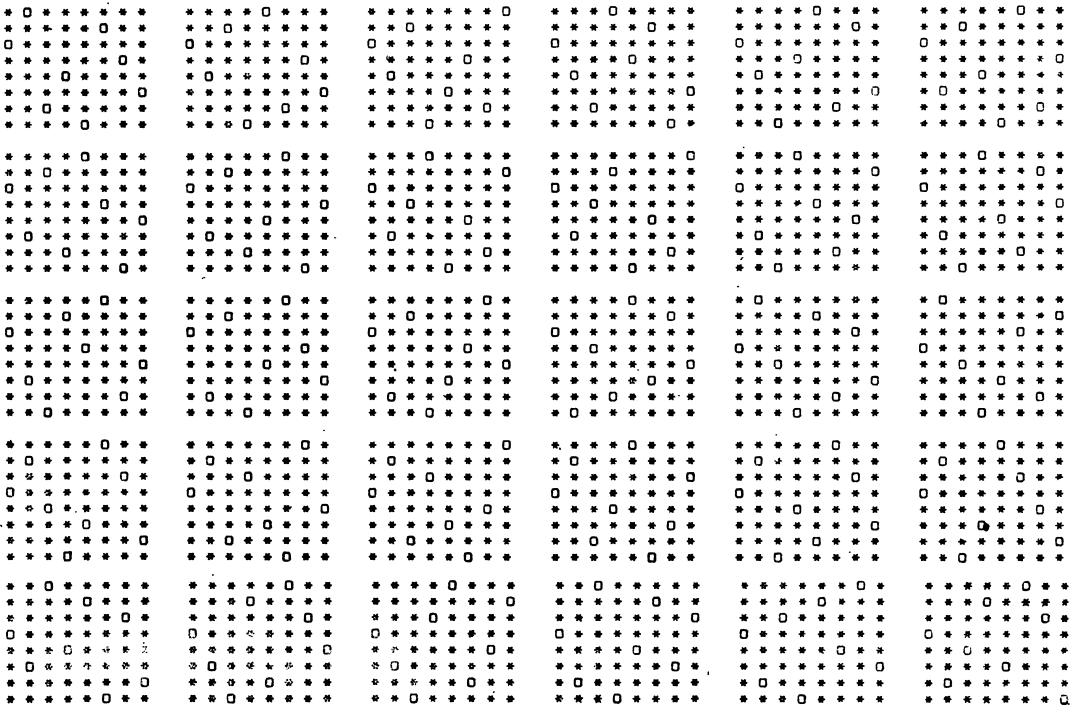
0.0.

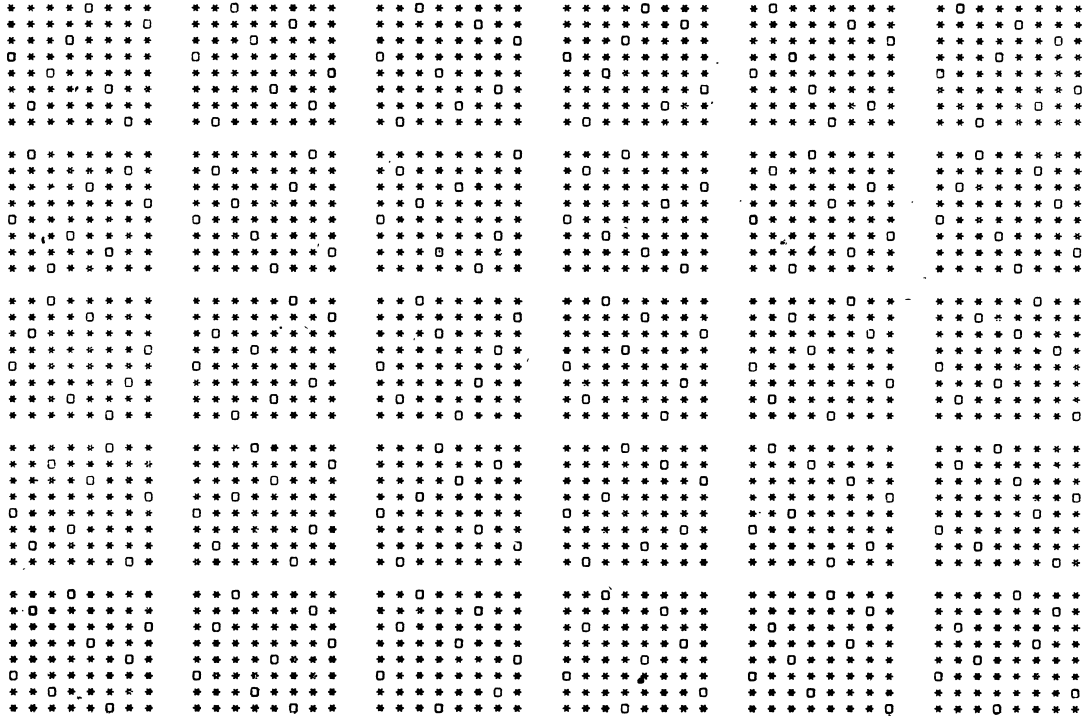
```

1.0 'BEGIN' 'INTEGER' N, N1, M;
1.1 СЛЕДУЮЩЕЕ N: N:=ININT; LINES PER PAGE(1000); OUTIMAGE; OUTTEXT('***РАССТАНОВКА ***'); OUTINT(N,4); OUTTEXT(' ' FERZEI
1.6 ***'); OUTIMAGE; N:=2*N+3; M:=ENTIER(120/N);
1.9 'COMMENT' M--ЧИСЛО БОСКОК В МАССИВЕ ВЫВОДА (РАССТАНОВКИ). M=ЧИСЛУ БОСКОК ВЫВОДИМЫХ НА ОДНОЙ СТРОКЕ;
2.0 'BEGIN' 'REF' (QUEEN) 'ARRAY' ФЕРЗИ[1:N], 'BOOLEAN' 'ARRAY' ГОР, ВЕРТ[1:N], ВОСХОД[2:2*N], НИСХОД[-N+1:N-1]; 'INTEGER' K, L
2.0 '
2.0 'COMMENT' K, L- РАБОЧИЕ ПЕРЕМЕННЫЕ,
2.0 'INTEGER' 'ARRAY' РАССТАНОВКИ[1:M,1:N]; 'INTEGER' НОМЕР БОСКИ;
3.0 'CLASS' QUEEN(НОМЕР, КЛЕТКА); 'INTEGER' НОМЕР, КЛЕТКА;
4.0 'BEGIN' ДЕТАСН;
4.1 'COMMENT' ПЕРЕХОД В САМОСТОЯТЕЛЬНОЕ СОСТОЯНИЕ, ЗАВЕРШЕНИЕ ГЕНЕРАЦИИ;
4.1 ПОИСКИ: 'COMMENT' ДВИГАЕМСЯ ПО ВЕРТИКАЛИ В ПОИСКАХ СВОБОДНОЙ КЛЕТКИ;
4.2 'IF' СВОБОДНА(НОМЕР, КЛЕТКА) 'THEN'
5.0 'BEGIN' ЗАНЯТЬ(НОМЕР, КЛЕТКА); 'IF' НОМЕР < N 'THEN' RESUME(ФЕРЗИ[НОМЕР + 1])
5.2 'ELSE' ЗАПОМНИТЬ РАССТАНОВКУ,
5.2 'COMMENT' ИШЕМ СЛЕДУЮЩУЮ СВОБОДНУЮ КЛЕТКУ, ОСВОБОЖДАЯ ЗАНЯТУЮ,
5.3 ОСВОБОДИ(НОМЕР, КЛЕТКА),
5.5 'END',
4.3 СОВИГ. КЛЕТКА:=КЛЕТКА + 1; 'IF' КЛЕТКА > N 'THEN'
5.0
6.0 'BEGIN' 'COMMENT' ВОЗВРАЩАЕМСЯ В ИСХОДНОЕ ПОЛОЖЕНИЕ И "ТОЛКАЕМ" ПРЕДЫДУЩЕГО ФЕРЗЯ. ЕСЛИ ТАКОВОГО
6.0 НЕ ОКАЗЫВАЕТСЯ (ТЕКУЩИЙ ФЕРЗЬ - ПЕРВЫЙ), ТО ЗАКАНЧИВАЕМ РАБОТУ, ВОЗВРАЩАЯ УПРАВЛЕНИЕ ГЛАВНОЙ ПРОГРА
6.0 М-МЕ (ОПЕРАТОР ДЕТАСН СМЕТКОМ ВСЕ);
6.1 КЛЕТКА := 1; 'IF' НОМЕР = 1 'THEN' ВСЕ: ДЕТАСН; RESUME(ФЕРЗИ[НОМЕР - 1]);
6.5 'END';
4.5 'GOTO' ПОИСКИ;
4.5 'COMMENT' ПРОДОЛЖАЕМ ПОИСК СВОБОДНОЙ КЛЕТКИ;
4.7 'END' QUEEN;
7.0 'BOOLEAN' 'PROCEDURE' СВОБОДНА(НОМЕР, КЛЕТКА); 'VALUE' НОМЕР, КЛЕТКА; (INTEGER' НОМЕР, КЛЕТКА; СВОБОДНА := 'NOT' (
7.1 ГОР[КЛЕТКА] 'OR' ВЕРТ[НОМЕР] 'OR' ВОСХОД[НОМЕР+КЛЕТКА]
7.1 'OR' НИСХОД[КЛЕТКА-НОМЕР]);
8.0 'PROCEDURE' ЗАНЯТЬ(НОМЕР, КЛЕТКА); 'VALUE' НОМЕР, КЛЕТКА; 'INTEGER' НОМЕР, КЛЕТКА; ГОР[КЛЕТКА]:=ВЕРТ[НОМЕР]. =ВОСХОД
8.1 [КЛЕТКА+НОМЕР]:=НИСХОД[КЛЕТКА-НОМЕР]
8.1 := 'TRUE';
9.0 'PROCEDURE' ОСВОБОДИ(НОМЕР, КЛЕТКА); 'VALUE' НОМЕР, КЛЕТКА; 'INTEGER' НОМЕР, КЛЕТКА; ГОР[КЛЕТКА]:=ВЕРТ[НОМЕР]:=ВОСХОД
9.1 0[КЛЕТКА+НОМЕР]:=НИСХОД[КЛЕТКА-НОМЕР]
9.1 := 'FALSE';
10.0 'PROCEDURE' ЗАПОМНИ РАССТАНОВКУ;
11.0 'BEGIN' 'IF' НОМЕР БОСКИ = M 'THEN'
12.0 'BEGIN' ПЕЧАТЬ;
12.1 'COMMENT' ЗАПОЛНЕННЫЙ МАССИВ ВЫВОДИТСЯ;

```









МАХ РАЗМЕР ПАМЯТИ, ЗАНЯТОЙ ПОД ПО (В БАЙТАХ) :  
1588

## 9.2. Программа класса SIMTAPL

СИМУЛА - 67. ВЕРСИЯ 3.1 ИМПАН СССР - МИИ. 18.03.83.

0.0 ИМЯ:BSIMTP,

1.0 'BEGIN' 'REAL' A;

2.0 SIMULATION 'CLASS' S I M T A P L;

3.0 'BEGIN'

4.0 PROCESS 'CLASS' P L O T (TITLE,N), 'TEXT' TITLE; 'INTEGER' N;

5.0 'BEGIN' 'INTEGER' NF,K; 'REF' (HEAD) LF, 'REF' (FUNCTION) TF; 'REF' (FUNCTION) YA, YF; LINK

6.0 'CLASS' FUNCTION(Y,C); 'NAME' Y, 'REAL' Y, 'CHARACTER' C;

7.0 'BEGIN' 'ARRAY' F(1:N); 'INTEGER' K; 'REAL' FMAX, FMIN, TFUN, 'REAL' CD;

7.1 FMIN:=10.0\$18; FMAX:=-10.0\$18;

7.4 'END' FUNCTION;

8.0 'PROCEDURE' FUNC(Y,C); 'NAME' Y, 'REAL' Y, 'CHARACTER' C;

9.0 'BEGIN' 'IF' NF = 0 'THEN'

10.0 'BEGIN' LF:='NEW' HEAD, 'IF' 'THIS' PLOT 'IS' T PLOT 'THEN' 'ACTIVATE' 'THIS' PLOT 'DELAY' 0.

10.4 'END';

9.2 NF := NF + 1, 'NEW' FUNCTION(Y,C). INTO(LF);

9.5 'END' ЗАПИСАНИЯ ФУНКЦИИ;

11.0 'PROCEDURE' T A B D E P(SF,SA); 'CHARACTER' SF,SA;

12.0 'BEGIN' 'ARRAY' X,Y(0:K); 'INTEGER' I;

12.1 ЗАГОЛОВОК('ТАБЛИЦА',SA,SF); SYSOUT. IMAGE. SETPOS(14); OUTCHAR(SA), SYSOUT. IMAGE. SETPOS,

12.4 S(34); OUTCHAR(SF), OUTIMAGE; НАДТИ(SA,SF);

12.7 'COMMENT' УПОРЯДОЧИВАЕМ X И Y ПО ВОЗРАСТАНИЮ X;

12.8 SORT(X,Y,K); 'FOR' I:=1 'STEP' 1 'UNTIL' K 'DO'

13.0 'BEGIN' OUTREAL(X[I],10,20); OUTREAL(Y[I],10,20); OUTIMAGE;

13.5 'END';

12.11 'END' T A B D E P;

14.0 'PROCEDURE' P L O T D E P(SF,SA); 'CHARACTER' SF,SA;

15.0 'BEGIN' 'ARRAY' X,Y(0:K); 'INTEGER' I; 'REAL' D,B, 'INTEGER' L,NL; 'TEXT' BL,

15.1 ЗАГОЛОВОК('ГРАФИК',SA,SF); НАДТИ(SA,SF); SORT(X,Y,K); OUTTEXT('

15.4 ' ДИАПАЗОНН '); OUTTEXT(' МИНИМУМ '); OUTTEXT(' МАКСИМУМ '); D

15.7 OUTTEXT(' ЦЕНА ДЕЛЕНИЯ'); OUTIMAGE; OUTTEXT(' ПО '); OUTCHAR(SA), OUTREAL(X[1],10,

15.11 22); OUTREAL(X[K],10,20); D:=(X[K]-X[1])/K; OUTREAL(D,10,19); OUTIMAGE; OUTTEXT('

15.16 ПО '); OUTCHAR(SF); OUTREAL(YF FMIN,10,22); OUTREAL(YF FMAX,10,20); YF CD:=(YF FMAX -

15.20 YF FMIN)/100; OUTREAL(YF CD,10,19); OUTIMAGE; OUTTEXT(' ЧИСЛО НАБЛЮДЕНИЙ='); OUTINT

15.24 (K,5); OUTIMAGE; ОСБ X; BLX-BLANKS(18); NL:=1, B:=X[1]+D, I:=1, 'FOR' I:=1 'STEP'

15.31 1 'UNTIL' K 'DO'

```

16.0 'BEGIN' 'IF' NL = 5 'THEN'
17.0 'BEGIN' NL:=1; OUTINT(L,18); OUTCHAR("-");
17.5 'END'
16.1 'ELSE'
18.0 'BEGIN' OUTTEXT(BL); OUTCHAR ( " "); NL := NL + 1;
18.5 'END';
16.2 OUTCHAR( SA ); C T P O K A: 'IF' X[I] > B 'THEN' 'GOTO' K O H C T P; T O 4 K A: SYSOUT. I
16.4 MAGE. SETPOS(20+(Y[I]-YF.FMIN)/YF.CD); OUTCHAR( S F); I := I + 1; 'IF' I 'LE' K 'THEN'
16.7 'GOTO' C T P O K A
16.7 'ELSE' 'GOTO' K O H C T P; K O H C T P: OUTIMAGE, B := B + D;
16.11 'END' УИЖ/А ПО СТРОКАМ;
15.32 K O H C T P: OUTIMAGE;
15.34 'END' P L O T D E P.
19.0 'PROCEDURE' U P D A T E;
20.0 'BEGIN' PASOTA: K:=K+1; 'IF' K>N 'THEN' 'GOTO' KOHEU; 'IF' 'THIS' PLOT 'IS' 'PLOT' 'THEN' 'THIS'
20.3 TPLOT. A[K]:=TIME; 'FOR' TFX-LF. FIRST, TF. SUC 'WHILE' TF/= 'NONE' 'DO'
20.4 'INSPECT' TF 'DO'
22.0 'BEGIN' TFUN:=Y; 'IF' TFUN>FMAX 'THEN' FMAX:=TFUN; 'IF' TFUN<FMIN 'THEN' FMIN:=TFUN; K:=K+1;
22.5 F[K]:=TFUN;
22.7 'END' ЗАПЕРА ОУНКУИЯ;
20.5 KOHEU
20.6 'END' U P D A T E;
23.0 'PROCEDURE' R E S E T;
24.0 'BEGIN' K := 0; 'FOR' TFX-LF. FIRST, TF. SUC 'WHILE' TF/= 'NONE' 'DO'
25.0 'BEGIN' TF FMIN:=10.0*18; TF FMAX:=-10.0*18; TF.K=0;
25.5 'END';
24.4 'END' R E S E T;
26.0 'PROCEDURE' SORT(X,Y,K); 'ARRAY' X,Y; 'INTEGER' K;
26.0 'COMMENT' ПЕРИПИСЬ Y, A F B X, YF, F B Y И СОРТИРУЕТ X И Y;
27.0 'BEGIN' 'INTEGER' I,J,NM,NP; 'REAL' MIN,S; 'FOR' I:=1 'STEP' 1 'UNTIL' K 'DO'
28.0 'BEGIN' X[I] := YA F[I]; Y[I] := YF. F[I];
28.4 'END' ПЕРЕПИСИ;
27.2 MIN:=X[I]; NM:=1, NP:=1, 'FOR' I:=1 'STEP' 1 'UNTIL' K-1 'DO'
29.0 'BEGIN' NM:=NP; 'FOR' J:=I 'STEP' 1 'UNTIL' K 'DO' 'IF' X[J] < MIN 'THEN'
30.0 'BEGIN' NM:=J, MIN:=X[J];
30.4 'END' ;
29.3 S:=X[NP]; X[NP]:=MIN; X[NM]:=S; S:=Y[NP]; Y[NP]:=Y[NM]; Y[NM]:=S; NP := NP + 1;
29.10 MIN:=X[NP];
29.12 'END';
27.7 'END' S O R T;
31.0 'PROCEDURE' H A M TI(SA,SF); 'CHARACTER' SA,SF;
32.0 'BEGIN' 'FOR' TFX-LF. FIRST, TF. SUC 'WHILE' TF/= 'NONE' 'DO' 'IF' TF.C = SA 'THEN' YAX-TF 'ELSE'
32.1 'IF' TF.C = SF 'THEN' YFX-TF, 'IF' K = 0 'THEN'

```

```

33.0 'BEGIN'OUTTEXT(''НЕТ ЗНАЧЕНИЯ ДЛЯ '');OUTCHAR(SA); OUTCHAR(SF); 'GOTO' КОЕЛУ
33.5 'END',
32 3 'IF' YA== 'NONE' 'THEN'
34 0 'BEGIN'OUTTEXT(''НЕТ АРГУМЕНТА ''); OUTCHAR(SA);
34.4 'END',
32 4 'IF' YF == 'NONE' 'THEN'
35 0 'BEGIN'OUTTEXT(''НЕТ ФУНКЦИИ ''); OUTCHAR(SF);
35.4 'END',
32.5 КОЕЛУ;
32.6 'END' НА И Т И;
36.0 'PROCEDURE' ЗАГОЛОВКА(Т, SA, SF); 'TEXT' Т; 'CHARACTER' SĀ, SF;
37.0 'BEGIN' 'TEXT' BL;
37.1 OUTIMAGE; OUTTEXT('' '); OUTTEXT(Т); OUTTEXT('' ЗАВИСИМОСТИ ''); OUTCHAR(SF);
37.6 OUTTEXT('' ОТ ''); OUTCHAR(SA); OUTCHAR(" "); OUTIMAGE;
37.11 'END' ЗАГОЛОВКА,
38.0 'PROCEDURE' ОСЬХ;
39.0 'BEGIN' 'INTEGER' I;
39.1 OUTTEXT('' '); 'FOR' I:=0 'STEP' 10 'UNTIL' 100 'DO' OUTINT(I, 10); OUTIMAGE; OUTTEXT('
39.4 ' '); 'FOR' I:=1 'STEP' 1 'UNTIL' 10 'DO' OUTTEXT('' I---+----'); OUTTEXT('
39.6 'I-->'); OUTIMAGE;
39.9 'END' ОСЬ Х;
5.1 ENTRY (PLOT, FUNC, TABDEP, PLOTDEP, UPDATE, RESET, (SIMTB));
5.3 'END' PLOT;
40.0 PLOT 'CLASS' T P L O T (Т); 'REAL' Т;
41.0 'BEGIN' 'ARRAY' A(1:N);
42.0 'PROCEDURE' OUTTABLE;
43.0 'BEGIN' 'INTEGER' I; 'TEXT' BL;
43.1 OUTIMAGE; OUTTEXT('' ТАБЛИЦА ЗНАЧЕНИЙ ГРАФИКА ''); OUTTEXT(TITLE); OUTCHAR(" "); 'IF' К=
43.5 0 'THEN'
44.0 'BEGIN' OUTTEXT('' ПУСТА. ''); 'GOTO' END
44.3 'END',
43.6 OUTIMAGE; BL%←BLANKS(18); OUTTEXT('' ВРЕМЯ ''); 'FOR' TF%←LF.FIRST, TF.SUC 'WHILE' TF≠/'N
43.9 ONE' 'DO'
45.0 'BEGIN' OUTTEXT(BL); OUTCHAR(TF.C);
45.4 'END' ШАПКИ ТАБЛИЦЫ;
43.10 OUTIMAGE; 'FOR' I:=1 'STEP' 1 'UNTIL' К 'DO'
46.0 'BEGIN' OUTREAL(A(I), 10, 19); 'FOR' TF%←LF.FIRST, TF.SUC 'WHILE' TF≠/'NONE' 'DO' OUTREAL(TF.F
46.2 [I], 10, 19); OUTIMAGE;
46.5 'END' ВЫДАЧИ ТАБЛИЦЫ;
43.12 END OUTIMAGE;
43.14 'END' OUTTABLE;

```

```

47 0  'PROCEDURE' OUTPLOT;
48 0  'BEGIN' 'INTEGER' I;
48 1  OUTIMAGE, OUTTEXT(' ', OUTTEXT(TITLE), OUTIMAGE, OUTTEXT(' ' ДИАПАЗОНЫ ' '), OUT
48 6  TEXT(' ' МИНИМУМ ' '); OUTTEXT(' ', МАКСИМУМ ' '); OUTTEXT(' ' ЦЕНА ДЕЛЕНИ
48 8  Я ' '); OUTIMAGE, OUTTEXT(' ' ПО ВРЕМЕНИ ' '), OUTREAL(A[I], 10, 19), 'IF' K > 0 'THEN' OUTREAL(
48 12 A[K], 10, 19), OUTIMAGE, 'FOR' TF%-LF FIRST, TF SUC 'WHILE' TF /= 'NONE' 'DO'
49 0  'BEGIN' OUTTEXT(' ' ПО ' '), OUTCHAR(TF C), OUTTEXT(' ' ' '), OUTREAL( TF FMIN, 10, 19
49 4  ), OUTREAL(TF FMAX, 10, 19), TF CD =ABS(TF FMAX - TF FMIN)/100 ; OUTREAL(TF CD, 10, 19), OUT
49 8  IMAGE;
49 10 'END' ВЫДАЧИ ДИАПАЗОНОВ;
48 14 'COMMENT' В Ы О А Ч А Г Р А Ф И К А;
48 15 ОСЬ X, 'IF' K=0 'THEN' 'GOTO' КОН; 'FOR' I =1 'STEP' 1 'UNTIL' K 'DO'
50 0  'BEGIN' OUTREAL(A[I], 10, 19), OUTCHAR("I"), 'FOR' TF%-LF FIRST, TF SUC 'WHILE' TF /= 'NONE' 'D
50 3  O'
51 0  'BEGIN' SYSOUT, IMAGE, S E T P O S(20 + (TF F[I] - TF FMIN)/TF CD), OUTCHAR(TF C);
51 4  'END';
50 4  OUTIMAGE;
50 6  'END' ПЕЧАТИ СТРОК ГРАФИКА;
48 18 КОН;
48 19 'END' И ЗАГОТОВКА;
41 1  P A B O T A 'IF' K < N 'THEN' U P D A T E, 'INNER', HOLD(T), 'GOTO' РАБОТА; ENTRY(OUTTABLE, OUTPLOT, TPLLOT,
41 5  (SIMTB));
41 7  'END' T P L O T;
3 2  'END' SIMTAPL;
1 1  ENTRY (SIMTAPL, (SIMTB));
1 3  'END'
9 3  'EQP'

```

### 9.3. Программа класса DISCONT

СИМУЛА - 67 ВЕРСИЯ 3.1 ИМПАН СССР - МИФИ 18 03 83.

00 ИМЯ БДИСКО.

10 'BEGIN' 'REAL' A, 'EXTERNAL' (SIMTB);

20 SIMTAPL 'CLASS' DISCONT (DT); 'REAL' DT;

30 'BEGIN' 'REF' (HEAD) CONDLIST, 'REF' (INTERS) CHECK, 'REF' (HEAD) INTLIST, 'REF' (INTEGRATION) INTMET;

40 'PROCESS' CLASS INTERS;

50 'BEGIN' 'REF' (WANT) TY, 'REF' (WANT) E, 'REF' (PROCESS) C;

51 M: 'IF' CONDLIST. EMPTY THEN 'XOY' PASSIVATE, C% - CURRENT, TY% - CONDLIST. LAST, ОЧУСЛ 'IF' TY /= 'NON

54 E' THEN

60 'BEGIN' E% - TY, TY % - TY PRED.

63 'INSPECT' E 'DU'

7.1. 'IF' B THEN

8.0 'BEGIN' E OUT, 'ACTIVATE' E P 'AFTER' C;

8.4 'END';

64 'GOTO' ОЧУСЛ;

6.6 'END' ПРОСМОТРА СПИСКА УСЛОВИЙ.

5.5 'IF' CONDLIST. EMPTY THEN 'GOTO' XOY, 'REACTIVATE' CHECK 'AFTER' CURRENT NEXTEV, 'GOTO' M,

59 'END' INTERS.

90 LINK 'CLASS' WANT (B, P), 'NAME' B, 'BOOLEAN' B, 'REF' (PROCESS) P;

91

100 'PROCEDURE' WAITUNTIL (B); 'NAME' B, 'BOOLEAN' B,

110 'BEGIN' 'NEW' WANT (B, CURRENT) INTO (CONDLIST), 'ACTIVATE' CHECK 'AFTER' CURRENT NEXTEV, PASSIVATE;

115 'END' WAITUNTIL;

120 'PROCEDURE' ACTCOND (X, B); 'NAME' B, 'BOOLEAN' B, 'REF' (PROCESS) X;

130 'BEGIN' 'NEW' WANT (B, X), INTO (CONDLIST), CANCEL (X), 'ACTIVATE' CHECK 'AFTER' CURRENT,

135 'END' ACTCOND;

```

14.0  PROCESS'CLASS'INTEGRATION.
      15 0  'BEGIN' 'REF' (INTGRL) I,
      15 1  PAGOTA HOLD(DT), 'IF' INTLIST EMPTY 'THEN'
          16.0  'BEGIN' XQY. PASSIVATE, 'GOTO' PAGOTA
          16.3  'END',
      15 3  'INNER', 'GOTO' PAGOTA;
      15 6  'END' INTEGRATION;
17 0  INTEGRATION'CLASS'EULIER;
      18 0  'BEGIN' 'FOR' I%-INTLIST FIRST, I. SUC 'WHILE' I != 'NONE' 'DO'
      18 1  'INSPECT' I 'DO'
          19.1  Y1 = Y + X * DT,
      18 2  'FOR' I%-INTLIST. FIRST, I. SUC 'WHILE' I != 'NONE' 'DO' I. Y' = I. Y1,
      18 4  'END' EULIER,
3 1  LINK
      20 0  'CLASS' INTGRL (Y, X), 'NAME' X; 'REAL' Y, X,
      20.0  'COMMENT' Y-ВЫХОД ИНТЕГРАТОРА, X-ЕГО ВХОДНАЯ ФУНКЦИЯ;
          21 0  'BEGIN' 'REAL' Y1,
          21 1  'THIS' INTGRL INTO (INTLIST); 'ACTIVATE' INTMET 'DELAY' 0,
          21 4  'END' INTGRL,
      3 1  ENTRY ( WAIT UNTIL, ACTCOND, EULIER, INTGRL, INTEGRATION, (DISCO)); CONDLIST%- 'NEW' HEAD, CHECK%- 'NEW' INTERS; INTL
      3 4  IST%- 'NEW' HEAD; INTMET%- 'NEW' EULIER, 'INNER',
      3 8  'END' DISCONT;
      1 1  ENTRY (DISCONT, (DISCO));
      1 3  'END'
0 3  'EDP'

```

#### 9.4. Программа использования классов SIMTAPL и DISCONT

СИМУЛА - 67. ВЕРСИЯ 3.1 ИПМ АН СССР - МИФИ. 18.03.83.

```

0 0 ИМЯ:ТЕСТ;
1 0 'BEGIN' 'REAL' T, EPS, ШАГ; 'INTEGER' KT; 'EXTERNAL' (SIMTB); 'EXTERNAL' (DISCO);
1 1 T = INREAL; EPS = INREAL; ШАГ = INREAL; KT = ININT; LINES PER PAGE (1000); OUTIMAGE; OUTTEXT(' ПЕРИОД ВОЗДЕЙСТВИЯ='');
1 8 OUTFIX(T, 2, 5); OUTTEXT(' EPS=''); OUTFIX(EPS, 2, 5); OUTTEXT(' ШАГ ИНТЕГРИРОВАНИЯ=''), OUTFIX(ШАГ, 3, 6); OUTTEXT('
1.13 КОЛ-ВО ТОЧЕК НА ГРАФИКЕ=''); OUTINT(KT, 4); OUTIMAGE; DISCONT(ШАГ)
2 0 'BEGIN' 'REF' (INTGRL) Z, V; 'REF' (TPLOT) ГРАФИК;
3.0 PROCESS CLASS ВОЗДЕЙСТВИЕ;
4.0 'BEGIN' M: Z.Y := Z.Y / 1.3, V.Y := V.Y / 2, HOLD(T); 'GOTO' M,
4.6 'END' ВОЗДЕЙСТВИЯ;
2.1 Z% = 'NEW' INTGRL(50, -0.2 * Z.Y - 0.5 * V.Y); V% = 'NEW' INTGRL(0, 0.5 * Z.Y); ГРАФИК% = 'NEW' TPLOT(' Z=F(T), V=F(T)', KT, 10 * ША
2.3 Г); ГРАФИК.FUNC(Z.Y, "Z"); ГРАФИК.FUNC(V.Y, "V"); 'ACTIVATE' 'NEW' ВОЗДЕЙСТВИЕ, WAIT UNTIL(ABS(Z.Y) < EPS), OUTTEXT(
2 8 ' ИСКОМОЕ ВРЕМЯ=''); OUTFIX(TIME, 3, 10); ГРАФИК. OUTPLOT; ГРАФИК PLOTDER("Z", "V"); ГРАФИК OUTTABLE, ГРАФИК. TABDE
2 13 P("Z", "V"),
2.15 'END';
1.18 'END'
0 3 'EOP'
РАЗМЕР ПАМЯТИ, ВЫДЕЛЕННОЙ ПОД ПО (В КИТАХ) :
40000
АДРЕС НАЧАЛА ПАМЯТИ : 0D03C0
АДРЕС КОНЦА ПАМЯТИ : 0D9FFC

```

ПЕРИОД ВОЗДЕЙСТВИЯ= 0.30 EPS= 0.10 ШАГ ИНТЕГРИРОВАНИЯ= 0.005 КОЛ-ВО ТОЧЕК НА ГРАФИКЕ= 40

ИСКОМОЕ ВРЕМЯ= 4.800

Z=F(T), V=F(T)

ДИАПАЗОНЫ	МИНИМУМ	МАКСИМУМ	ЦЕНА БЕЛЕНИЯ
ПО ВРЕМЕНИ	0.000000E+00	1.9499840E+00	
ПО Z	4.1350280E+00	3.8461540E+01	3.4326500E-01
ПО V	0.0000000E+00	6.7590660E+00	6.7590650E-02



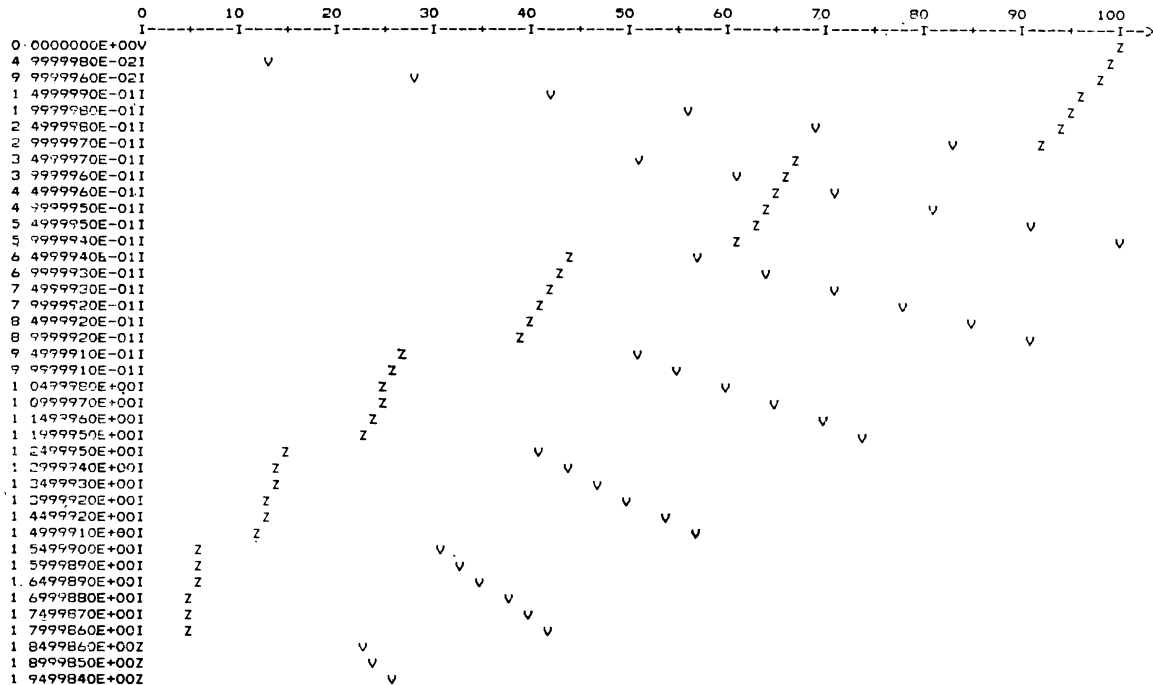


ГРАФИК ЗАВИСИМОСТИ Z ОТ V  
 ДИАПАЗОНЪ.

МИНИМУМ

МАКСИМУМ

ЦЕНА ДЕЛЕНИЯ

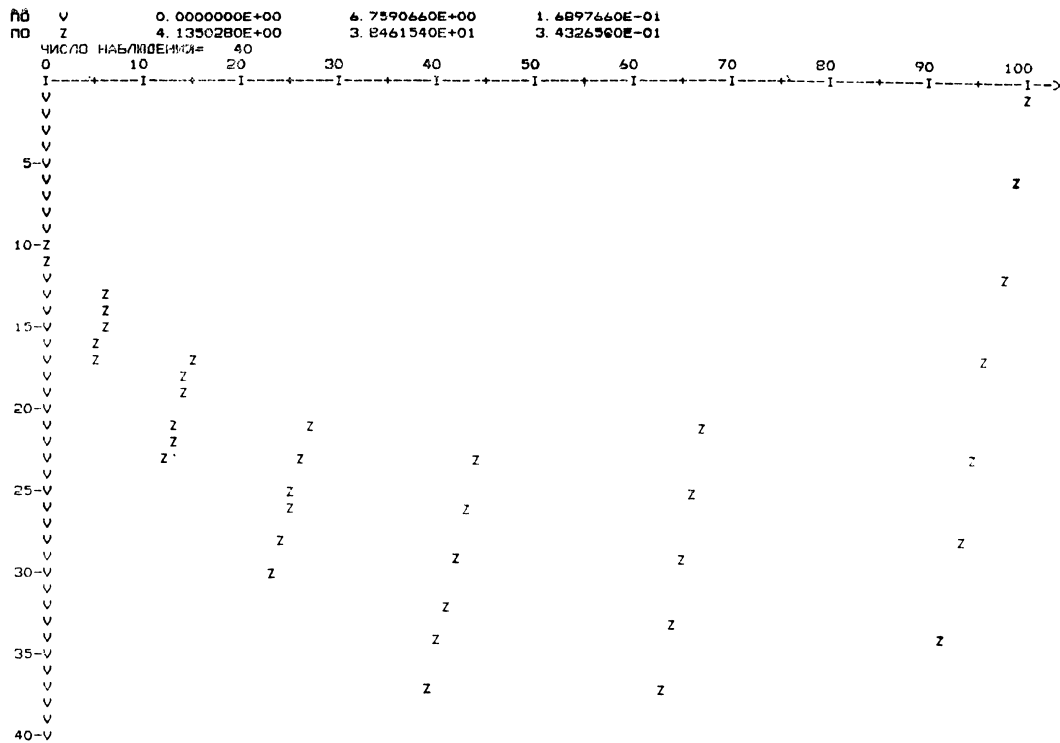


ТАБЛИЦА ЗНАЧЕНИЙ ГРАФИКА "  $Z=F(T)$ ,  $V=F(T)$  "

ВРЕМЯ	Z	V
0.0000000E+00	3 8461340E+01	0 0000000E+00
4.9999980E-02	3 8108100E+01	8 6187980E-01
9.9999960E-02	3 7654340E+01	1 9042300E+00
1.4999990E-01	3 7221490E+01	2 8407520E+00
1.9999980E-01	3 6769770E+01	3 7662410E+00
2.4999980E-01	3 6299660E+01	4 6802300E+00
2.9999970E-01	3 5811590E+01	5 5822620E+00
3.4999970E-01	2 7195920E+01	3 4758850E+00
3.9999960E-01	2 6831000E+01	4 1517000E+00
4.4999960E-01	2 6453000E+01	4 3182430E+00
4.9999950E-01	2.6062300E+01	5 4751940E+00
5.4999950E-01	2 5659210E+01	6 1222370E+00
5.9999940E-01	2.5244150E+01	6 7590660E+00
6.4999940E-01	1.9135680E+01	3 8618280E+00
6.9999930E-01	1.8843650E+01	4 3369490E+00
7.4999930E-01	1.8542800E+01	4 8046690E+00
7.9999920E-01	1.8233360E+01	5 2647700E+00
8.4999920E-01	1 7915670E+01	5 7170420E+00
8.9999920E-01	1 7539980E+01	6 1612810E+00
9.4999910E-01	1 3315600E+01	3 4164980E+00
9.9999910E-01	1 3094290E+01	3 7469050E+00
1.0499980E+00	1 2867030E+01	4 0717120E+00
1.0999970E+00	1 2634000E+01	4 3907740E+00
1.1499960E+00	1.2395420E+01	4 7039480E+00
1.1999950E+00	1.2151490E+01	5 0110940E+00
1.2499950E+00	9 1892830E+00	2 7374560E+00
1.2999940E+00	9 0271040E+00	2 9653670E+00
1.3499930E+00	8 8609130E+00	3 1891730E+00
1.3999920E+00	8 6908530E+00	3 4087910E+00
1.4499920E+00	8.5170680E+00	3 6241090E+00
1.4999910E+00	8 3397040E+00	3 8350410E+00
1.5499900E+00	6 3017790E+00	2 0766230E+00
1.5999890E+00	6 1856000E+00	2 2328610E+00
1.6199890E+00	6 0667200E+00	2 3861680E+00
1.6799880E+00	5.9452430E+00	2 5364680E+00
1.7499870E+00	5.8212670E+00	2 6837040E+00
1.7999860E+00	5.6948950E+00	2 8278160E+00
1.8499860E+00	4 3006640E+00	1 5225240E+00
1.8999850E+00	4 2197530E+00	1 6291160E+00
1.9499840E+00	4.1350280E+00	1 7336440E+00

ТАБЛИЦА ЗАВИСИМОСТИ Z ОТ V.

V	Z
0 0000000E+00	3. 8461540E+01
8 6187980E-01	3 8108100E+01
1 5225240E+00	4 3006640E+00
1. 6291180E+00	4 2187530E+00
1. 7336440E+00	4 1350280E+00
1. 9042300E+00	3 7654340E+01
2 0766230E+00	6 3017790E+00
2. 2328610E+00	6 1856000E+00
2. 3861660E+00	6 0667200E+00
2. 5364680E+00	5 9452430E+00
2. 6837040E+00	5 8212670E+00
2. 7374560E+00	9 1892830E+00
2. 8278160E+00	5 6948950E+00
2. 8407520E+00	3 7221490E+01
2. 9653670E+00	9 0271040E+00
3. 1891780E+00	8 8609130E+00
3. 4087910E+00	8. 6908530E+00
3. 4164980E+00	1. 3315600E+01
3. 4758850E+00	2. 7195920E+01
3. 6241090E+00	8. 5170680E+00
3. 7469050E+00	1. 3094290E+01
3 7662410E+00	3. 6769770E+01
3 8350410E+00	8 3397040E+00
3 8616280E+00	1 9125680E+01
4. 0717120E+00	1 2867030E+01
4. 1517000E+00	2 6831000E+01
4 3369490E+00	1 8843650E+01
4. 3907740E+00	1 2634000E+01
4 6802300E+00	3 6299660E+01
4 7039480E+00	1 2395420E+01
4 8046690E+00	1 8542800E+01
4 8182430E+00	2 6453000E+01
5 0110940E+00	1 2151490E+01
5 2647700E+00	1 9233360E+01
5 4751940E+00	2 6062300E+01
5 5822620E+00	3 5811590E+01
5 7170420E+00	1 7915670E+01
6 1222370E+00	2 5659210E+01
6 1612810E+00	1 7589980E+01
6 7570660E+00	2 5244150E+01

КОНЕЦ РАБОТЫ ПРОГРАММЫ

МАХ РАЗМЕР ПАМЯТИ, ЗАНЯТОЙ ПОД ПО (В БАЙТАХ) 3752

## СПИСОК ЛИТЕРАТУРЫ

1. Андрианов А. Н., Бычков С. П., Лезина И. М., Хорошилов А. И. Компилятор с языка симула-67 для БЭСМ-6.— М., 1980.— 51 с. (Препринт/ИПМ им. М. В. Келдыша АН СССР).
2. Андрианов А. Н., Бычков С. П., Хорошилов А. И. Средства раздельной компиляции симула-программ на БЭСМ-6 и ЕС ЭВМ.— М., 1982.— 40 с. (Препринт/ИПМ им. М. В. Келдыша АН СССР).
3. Андрианов А. Н., Задыхайло И. Б. Некоторые особенности программирования на ЭВМ CRAY-1.— М., 1982.— 24 с. (Препринт/ИПМ им. М. В. Келдыша АН СССР).
4. Андрианов А. Н., Задыхайло И. Б. Имитационное моделирование одной вычислительной системы.— В кн.: Моделирование дискретных управляющих и вычислительных систем: Тез. докл. 3 Всесоюзного семинара. Свердловск: Изд-во УНЦ АН СССР, 1981, с. 116—117.
5. Базисный РЕФАЛ. Описание языка и основные приемы программирования (методические рекомендации). Вып. V-33.— М.: ЦНИПИАСС, 1974.— 95 с.
6. Бычков С. П., Красовский А. Г. Об интерфейсе стенда моделирования и ПО бортовых ЦВМ.— В кн.: Моделирование дискретных управляющих и вычислительных систем: Тез. докл. 3 Всесоюзного семинара. Свердловск: Изд-во УНЦ АН СССР, 1981, с. 124—125.
7. Бычков С. П., Попкова Г. В., Хорошилов А. И. Компилятор с языка симула-67 для ЕС ЭВМ.— М., 1982.— 42 с. (Препринт/ИПМ им. М. В. Келдыша АН СССР).
8. Глушков В. М., Гусев В. В., Марьянович Т. П., Сахнюк М. А. Программные средства моделирования непрерывно дискретных систем.— Киев: Наукова думка, 1975.
9. Гродано П. Программирование на языке Паскаль/Пер. с англ. под ред. Подшивалова Д. Б.— М.: Мир, 1982.— 384 с.
10. Дал У. И., Мюрхауг Б., Нюгорд К. Симула-67. Универсальный язык программирования.— М.: Мир, 1969.— 99 с.
11. Дейкстра Э. Заметки по структурному программированию.— В кн.: Структурное программирование/Пер. с англ. под ред. Э. З. Любимского и В. В. Мартынюка.— М.: Мир, 1975, с. 7—97.
12. Дрожжинов В. И., Морозова Н. Д. Имитационная модель сети ЭВМ СЕКОП.— В кн.: Моделирование дискретных управляющих и вычислительных систем: Тез. докл. 3 Всесоюзного семинара. Свердловск: Изд-во УНЦ АН СССР, 1981, с. 118—120.

13. Ерофеев В. И., Меркушов Ю. П., Першиков В. И., Соколов А. П. Средства отладки программ в ОС ЕС ЭВМ. Справочное пособие/Под ред. В. Н. Лебедева.— М.: Статистика, 1979.— 245 с.
14. Климов Ан. В., Романенко С. А. Рефал в мониторинговой системе Дубна. Интерфейс рефала и фортрана.— М., 1975.— 34 с. (Препринт/ИПМ им. М. В. Келдыша АН СССР).
15. Кузин Л. Т., Бычков С. П., Храмов А. А. Представление понятийных знаний с помощью языка симула-67.— В кн.: Проектирование интеллектуальных систем. М.: Атомиздат, 1980, с. 54—61.
16. Лебедев В. М., Соколов А. П. Введение в систему программирования ОС ЕС.— М.: Статистика, 1978.— 144 с.
17. Лесюк В. Г., Марков А. С., Пеледов Г. В., Райков Л. Д. Система математического обеспечения ЕС ЭВМ.— М.: Статистика, 1974.— 216 с.
18. Мазный Г. Л. Программирование на БЭСМ-6 в системе Дубна.— М.: Наука, 1978.— 272 с.
19. Марьянович Т. П., Азаров С. С., Гусев В. В. и др. Имитационное моделирование средствами системы НЕДИС и gasp-IV.— Кибернетика, 1980, № 3, с. 35—50.
20. Морозова Л. Б. Транслятор с автокода БЕМШ в мониторинговой системе Дубна.— М., 1973.— 26 с. (Препринт/ИПМ им. М. В. Келдыша АН СССР).
21. Павловский В. Е. Моделирование вычислительных процессов в системе управления шагающим роботом.— В кн.: Робототехнические системы в отраслях народного хозяйства: Тез. докл. 2 Всесоюзного совещания. Минск, 1981, ч. 3, с. 147—148.
22. Радд У. Программирование на языке ассемблера и вычислительные машины системы IBM 360 и 370/Пер. с англ. под ред. Л. Д. Райкова.— М.: Мир, 1979.— 591 с.
23. Родионов А. С. Некоторые вопросы применения языка симула-67 для разработки специализированного программного обеспечения.— В кн.: Системный анализ и исследование операций. Новосибирск: Изд-во ВЦ СО АН СССР, 1979, с. 23—30.
24. Родионов А. С. Реализация функций блока ADVANCE в классе ССМО языка симула-67.— В кн.: Математические и имитационные модели сложных систем СМ-6. Сборник трудов. Новосибирск: Изд-во ВЦ СО АН СССР, 1981, с. 79—88.
25. Салтыков А. И., Макаренко Г. И. Программирование на языке фортран для БЭСМ-6.— М.: Наука, 1984.— 272 с.
26. Симула. Описание эталонного языка.— М.: Изд-во ЦЭМИ АН СССР, 1976.— 190 с.
27. Система математического обеспечения ЕС ЭВМ/Под ред. А. М. Ларионова.— М.: Статистика, 1974.— 216 с.
28. Скотт Р., Сондак Н. ПЛ/1 для программистов/Пер. с англ.— М.: Статистика, 1977.
29. Хусаинов Б. С. Программирование ввода-вывода в ОС ЕС ЭВМ на языке ассемблера.— М.: Статистика, 1980.— 264 с.

30. Шеннон Р. Имитационное моделирование систем — искусство и наука.— М.: Мир, 1978.— 418 с.
31. Шрайбер Т. Дж. Моделирование на GPSS.— М.: Машиностроение, 1980.— 589 с.
32. Штаркман В. С. Сравнительный анализ транслятора FOREX.— М., 1978.— 56 с. (Препринт/ИПМ им. М. В. Келдыша АН СССР: 129 М.)
33. Яковлев Е. И. Машинная имитация.— М.: Наука, 1975.— 158 с.
34. Birtwistle G. M. Discrete Event Modelling on SIMULA.— London: The MacMillon Press, 1979.
35. Birtwistle G. M. et all. SIMULA begin.— Philadelphia: Auerbach Publishers, 1973.
36. Franta W. R. The Process View of simulation.— New York: North-Holland, 1977.— 243 p.
37. Helsgaun K. DISCO — A Simula based language for continuous, combined and discrete simulation.— SIMULATION, 1980, N 7, p. 1—12.
38. Landwehr C. E. An abstract type for statistic collection in SIMULA.— ACM Transaction on Programming Languages and Systems, 1980, 2, N 4.
39. Makila K. CODASYL — type date base management system in SIMULA.— Management datamatics, 1976, 5, p. 113—121.
40. Palme J. Making SIMULA into a programming language for real time.— Management datamatics, 1975, 4, № 4, p. 129—137.
41. Palme J. Experience from the standartization of the SIMULA programming language.— Software — practice and experience, 1976, 6, N 3, pp. 405—409.
42. Palme J. Putting statistics into a SIMULA programm.— Management datamatics, 1976, 5, № 2, p. 79—82.
43. Prisker A. A. Fundamental concepts of GASP-IV. Proceedings of the International Symposium SIMULATION'77/Ed. M. H. Hamza. Montreux, June 22—24.— Zurich: Acta Press, 1977, p. 8—17.
44. Sol H. G. SIMULA(TION) in the analysis and désign of information systems. Proceedings of the International Symposium SIMULATION'77/Ed. M. H. Hamza. Montreux, June 22—24.— Zurich: Acta Press, 1977, p. 67—71.
45. Stormar E. PROSIM — A Toll for the real time programs.— SIMULA newsletter, 2, N 4, 1974, p. 3—8.
46. Under B. W., Pasker J. B. An operating System implementation and simulation language (OASIS)..In: Conference on Simulation measurement and modelling of computer systems.— New York, 1979, p. 151—164.

ББК 22.18

А 65

УДК 519.6

Андреанов А. Н., Бычков С. П., Хорошилов А. И. Программирование на языке симула-67.— М.: Наука. Глав. ред. физ.-мат. лит., 1985.— 288 с.

В книге рассматриваются на неформальном уровне основные средства языка симула-67, приводятся многочисленные примеры их применения. Особое внимание уделяется средствам имитационного моделирования и методике их применения при разработке моделей дискретных систем. Подробно рассматриваются работа с симула-программами на БЭСМ-6 и ЕС ЭВМ и применение реализованных в отечественных трансляторах средств раздельной компиляции, средств пакетной и диалоговой отладки, аппарата связи с программами на других языках.

Библиогр. 46 назв.

*Александр Николаевич Андреанов*

*Сергей Павлович Бычков*

*Александр Иванович Хорошилов*

#### ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИМУЛА-67

Серия: «Библиотечка программиста»

Редакторы *И. Б. Задыхайло, Л. Г. Силкова*

Художественный редактор *Т. И. Кольченко*

Техн. редактор *В. Н. Кондакова*

Корректоры *Л. И. Назарова, М. Л. Медведская*

ИБ № 12580

Сдано в набор 25.06.84. Подписано к печати 17.04.85. Т-07462. Формат 84×108<sup>1</sup>/<sub>32</sub>. Бумага тип. № 3. Гарнитура обыкновенная новая. Печать высокая. Усл. печ. л. 15,12. Усл. кр.-отт. 15,33. Уч.-изд. л. 16,66. Тираж 16 400 экз. Заказ № 280. Цена 1 р. 10 к.

Ордена Трудового Красного Знамени издательство «Наука»  
Главная редакция физико-математической литературы  
117071 Москва В-71, Ленинский проспект, 15

4-я типография издательства «Наука»  
630077 Новосибирск 77, Станиславского, 25

А  $\frac{1702070000-082}{053(02)-85}$  38-85

© Издательство «Наука».  
Главная редакция  
физико-математической  
литературы, 1985