В.О. САФОНОВ

Языки и методы программирования в системе ЭЛЬБРУС

681.3.06
С-21

В.О. САФОНОВ

Языки и методы программирования в системе «Эльбрус»

Под редакцией *С.С. Лаврова*

354032

МОСКВА "НАУКА"
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1989

ББК 22.18
С 21
УДК 519.68

Сафонов В.О. **Языки и методы программирования в системе "Эльбрус"**/Под ред. С.С. Лаврова. – М.: Наука. Гл. ред. физ.-мат. лит., 1989. – 392 с. – ISBN 5-02-013983-1.

Рассмотрены методы программирования на языках Эль-76, Паскаль, Клу и на других языках, реализованных на новых отечественных вычислительных комплексах "Эльбрус" с языко-ориентированной архитектурой. Описаны новые технологические принципы программирования, основанные на концепции структурированного интерфейса (ТИП-технология), опыт их применения и программная поддержка на МВК "Эльбрус". Приведены практические сведения для использования рассмотренных систем программирования. Дан анализ средств поддержки языков высокого уровня различных классов в архитектуре системы "Эльбрус", описаны новые методы реализации этих языков для МВК "Эльбрус".

Для научных работников и инженеров – специалистов по программному обеспечению ЭВМ, а также для студентов и аспирантов соответствующих специальностей.

Табл. 3. Ил. 15. Библиогр. 89 назв.

Рецензент член-корреспондент АН СССР *Б.А. Бабаян*

С $\frac{1404000000-077}{053(02)-89}$ 148-89

ISBN 5-02-013983-1

© Издательство "Наука".
Главная редакция
физико-математической литературы
1989

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ 5

ВВЕДЕНИЕ 9

В.1. Ориентация на языки высокого уровня 9
В.2. Система программирования МВК "Эльбрус" 17

Глава 1. **МЕТОДЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ЭЛЬ-76** 25

1.1. Пример программы 26
1.2. Классификация и обзор типов данных 29
1.3. Общие операции 30
1.4. Наборы 31
1.5. Числа 34
1.6. Векторы 37
1.7. Многомерные массивы 49
1.8. Особенности структуры программы 53
1.9. Описания 54
1.10. Классификация предложений 58
1.11. Последовательное и замкнутое предложение 60
1.12. Условное предложение 61
1.13. Выбирающее предложение 62
1.14. Циклы 63
1.15. Структурное предложение 65
1.16. Процедуры 72
1.17. Текстовые макросы 77
1.18. Программа 80
1.19. Форматный обмен 83
1.20. Основные понятия системы простых файлов 88
1.21. Файл: создание, открытие, закрытие, атрибуты 91
1.22. Контейнер: основные операции и атрибуты 95
1.23. Справочники. Архив 96
1.24. Обмен с барабанами и дисками 103
1.25. Обмен с лентами 107
1.26. Терминальные файлы. Диалог 110
1.27. Текстовые файлы 115
1.28. Ошибочные ситуации при обмене 118
1.29. Некоторые особенности семантики СПФ 119
1.30. Взаимодействие с операционной системой 121
1.31. Средства модульного программирования 129
1.32. Определяемый синтаксис 135
1.33. Язык стандартного диалога 138

Глава 2. СИСТЕМА ПРОГРАММИРОВАНИЯ ПАСКАЛЬ-ЭЛЬБРУС 151
2.1. Эволюция языка Паскаль 151
2.2. Принципы разработки системы Паскаль-Эльбрус 154
2.3. Особенности входного языка Паскаль-Эльбрус 155
2.4. Методы программирования и управления транслятором 164
2.5. Исполнение и отладка паскаль-программ 178

Глава 3. ЯЗЫК ПРОГРАММИРОВАНИЯ КЛУ И СИСТЕМА КЛУ-ЭЛЬБРУС . 189
3.1. Практические аспекты абстрактных типов данных 189
3.2. Обзор основных понятий языка Клу 194
3.3. Встроенные типы и генераторы типов 197
3.4. Типы, объекты, константы и переменные 207
3.5. Управляющие конструкции 210
3.6. Процедуры 215
3.7. Итераторы 217
3.8. Кластеры 219
3.9. Параметризованные модули 225
3.10. Библиотека модулей 230
3.11. Система программирования Клу-Эльбрус 231

Глава 4. СПЕЦИАЛИЗИРОВАННЫЕ СИСТЕМЫ ПРОГРАММИРОВАНИЯ ДЛЯ МВК "ЭЛЬБРУС" 243
4.1. Расширяемый язык АБВ и система АБВ-Эльбрус 244
4.2. Система Снобол-Эльбрус 252
4.3. Система Рефал-Эльбрус 255
4.4. Диалоговая система ДИАШАГ 258
4.5. Диалоговая система Форт-Эльбрус 262
4.6. Система Модула-2-Эльбрус 264

Глава 5. ТИП-ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ 267
5.1. Общий подход к технологиям программирования 268
5.2. Обзор технологий программирования 268
5.3. Принципы ТИП-технологии 271
5.4. Методы разработки программ по ТИП-технологии 277
5.5. Применение ТИП-технологии в трансляторах 302
5.6. Развитие ТИП-технологии 307

Глава 6. МЕТОДЫ РЕАЛИЗАЦИИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ В СИСТЕМЕ "ЭЛЬБРУС" 311
6.1. Архитектура МВК "Эльбрус" с точки зрения разработчика трансляторов 311
6.2. Новые методы реализации 336

Приложение 1. СИНТАКСИС СТАНДАРТНОГО ЯЗЫКА ПАСКАЛЬ И ЯЗЫКА ПАСКАЛЬ-ЭЛЬБРУС 352

Приложение 2. РУССКАЯ ЛЕКСИКА ЯЗЫКА ПАСКАЛЬ-ЭЛЬБРУС 359

Приложение 3. СИНТАКСИС ЯЗЫКА КЛУ И ЯЗЫКА КЛУ-ЭЛЬБРУС 361

Приложение 4. РУССКАЯ ЛЕКСИКА И ТЕРМИНОЛОГИЯ ЯЗЫКА КЛУ . . . 367

Приложение 5. СИНТАКСИС ЯЗЫКА АБВ И ОСОБЕННОСТИ ЯЗЫКА АБВ-ЭЛЬБРУС 373

СПИСОК ЛИТЕРАТУРЫ 386

ПРЕДИСЛОВИЕ

Система "Эльбрус" [9, 52] – это отечественная серия многопроцессорных вычислительных комплексов (МВК) с архитектурой, ориентированной на языки высокого уровня. В настоящее время существуют две модели (МВК "Эльбрус-1" и МВК "Эльбрус-2") и создаются новые модели ЭВМ этой серии.

Основная отличительная черта системы "Эльбрус" для программиста – удобство программирования на языках высокого уровня. Языки ассемблерного класса в этой системе отсутствуют. Ее базовый язык Эль-76 [52], на котором написано общее системное программное обеспечение (ОСПО) МВК "Эльбрус", не уступает по мощности современным языкам программирования, выполняет в системе универсальные функции и имеет эффективную реализацию, основанную на аппаратной поддержке базовых конструкций (процедура, массив, цикл, ситуация, модуль и др.). Для системы "Эльбрус" реализованы практически все широко используемые и некоторые новые языки.

В системе "Эльбрус" программист на всех этапах создания программы, включая отладку, управление заданиями и файлами, работает только на языках высокого уровня, не "опускаясь" до машинных понятий, таких как регистр, шестнадцатиричная распечатка памяти, адрес прерывания. Аппаратура и операционная система (ОС) обеспечивают защиту от наиболее распространенных видов ошибок, например неверной адресации или использования неопределенного значения переменной.

Инструментарий системного программиста на МВК "Эльбрус" включает:

– базовый язык высокого уровня Эль-76, на котором программируются как функциональные модули, так и операции управления задачами, процессами, файлами, архивом, обращения к ОС и другим компонентам ОСПО;

– технологические пакеты для поддержки разработки трансляторов;

– интерфейс ОС;

— многоязыковые компоненты ОСПО, единые для всех систем программирования: комплексатор, систему динамической диагностики, символьный отладчик, систему получения документации на текст программы.

Все эти факторы, как показал более чем десятилетний опыт разработки программного обеспечения МВК "Эльбрус" и эксплуатации системы, способствует удобству программирования и высокой производительности труда как системных, так и прикладных программистов. По оценкам пользователей, производительность труда программиста на МВК "Эльбрус", по сравнению с применением более традиционных ЭВМ для решения аналогичных задач, повышается примерно на порядок. Это подтверждается и личным опытом автора. Кроме того, система "Эльбрус", которая задумана как семейство ЭВМ универсального назначения, позволила решить многие задачи, которые из-за различных ограничений (по времени, памяти, динамике исполнения) не могли быть решены на других ЭВМ.

Благодаря развитой языко-ориентированной архитектуре и системному программному обеспечению (уже сейчас, через несколько лет после появления первой модели МВК, сравнимому по объему с программным обеспечением БЭСМ-6 и других популярных вычислительных систем), система "Эльбрус" освобождает программиста от решения ряда рутинных задач и тем самым стимулирует его творческие возможности. Это подтверждает опыт и результаты многих коллективов специалистов, занимающихся разработкой программного обеспечения МВК "Эльбрус": Института точной механики и вычислительной техники АН СССР (архитектура МВК, операционная система, система файлов, система программирования Эль-76, многоязыковые компоненты ОСПО, технологические пакеты); Новосибирского филиала ИТМиВТ (системы программирования Фортран, Кобол, ПЛ-1, Алгол, пакеты прикладных программ, средства машинной графики); Ленинградского университета (системы программирования: Паскаль, КЛУ, АБВ, Рефал, Снобол-4, ДИАШАГ, Форт, комплекс пакетов прикладных программ); Института кибернетики АН ЭССР (интеллектуальная система программирования МИС, система программирования Лисп); Ростовского университета (система программирования Симула-67) и многих других. Следует подчеркнуть, что входные языки системы программирования МВК "Эльбрус" (Фортран, Алгол, Паскаль и др.) учитывают сложившиеся диалекты для распространенных ЭВМ и, следовательно, обеспечивают преемственность для пакетов программ, написанных на этих языках программирования и функционирующих на других типах ЭВМ.

Данная книга посвящена языковым аспектам системы "Эльбрус" и является практическим руководством по программированию в системах Эль-76 (гл. 1), Паскаль-Эльбрус (гл. 2), Клу-Эльбрус (гл. 3), АБВ-Эльбрус, Снобол-Эльбрус, Рефал-Эльбрус, ДИАШАГ, Модула 2-Эльбрус (гл. 4). В главе 5 описаны новые технологические принципы программирования, используемые при разработке трансляторов для МВК "Эльбрус" в Ленинградском университете. Главы 1–5 предназначены для широкого круга программистов, желающих изучить методы программирования для МВК "Эльбрус"

на различных языках. Описание особенностей языков Клу и АБВ, еще недостаточно широко известных, представляет и самостоятельный интерес. Глава 6 носит более специальный характер. В ней рассматриваются архитектура МВК "Эльбрус" с точки зрения системного программиста – разработчика трансляторов и некоторые новые методы реализации, основанные на использовании возможностей аппаратуры МВК "Эльбрус".

В книге отражен десятилетний практический опыт и научные результаты автора в области разработки систем программирования МВК "Эльбрус". Использован также опыт преподавания принципов системы "Эльбрус" и языка Эль-76 на математико-механическом факультете ЛГУ. В частности, гл. 1 можно рассматривать как дальнейшее развитие учебника по языку Эль-76 [60], с учетом эволюции системы программирования Эль-76 и пожеланий пользователей МВК "Эльбрус" о публикации более полного практического руководства по программированию на языке Эль-76. Однако в этом отношении книга не претендует на исчерпывающую полноту, так как ввиду многообразия возможностей языка Эль-76 полный учебник по нему занял бы, по-видимому, несколько томов. Эталонное описание языка Эль-76 дано в работе его автора В.М. Пентковского [52].

Разумеется, в рамках одной книги невозможно было охватить и многие другие аспекты системы: организацию аппаратуры МВК "Эльбрус"; ее сравнение с традиционными ЭВМ; организацию и функционирование ОСПО; системы программирования Фортран, Алгол, Алгол-68, Симула-67, Кобол, ПЛ/1; пакеты прикладных программ и многие другие вопросы. Обзор имеющейся литературы по системе "Эльбрус" дан в работе [21].

В книге нашел отражение огромный труд коллектива сотрудников сектора математического обеспечения МВК "Эльбрус" НИИ математики и механики, преподавателей и аспирантов кафедры математического обеспечения ЭВМ ЛГУ: С.В. Вдовкина – основного разработчика систем программирования Паскаль и Клу, моего надежного друга и помощника; А.А. Кубенского – активного участника разработки системы АБВ, Паскаль и Клу, основного разработчика системы Рефал; В.Н. Рябинина и В.Ю. Самойлова – создателей системы программирования ДИАШАГ; Н.Н. Болдиновой и А.Н. Смертина – разработчиков системы Снобол-4; М.В. Дмитриевой и В.А. Лаврищевой – авторов системы Рефал; А.Е. Соловьева – разработчика системы Форт; А.А. Поспелова – разработчика системы Модула-2. В работах принимали участие Н.В. Москвина, М.А. Плаксин, С.В. Сафонова, Л.Б. Соколинский и И.А. Суслина, а также многие студенты кафедры. Всем им я глубоко благодарен за постоянное сотрудничество, ценные обсуждения и поддержку. Их работа способствовала и написанию данной книги.

Особенно я благодарен редактору книги, моему учителю программирования С.С. Лаврову. Именно по инициативе Святослава Сергеевича в 1977 г. в ЛГУ были начаты работы по программному обеспечению МВК "Эльбрус", и под его руководством наша небольшая группа студентов превратилась в коллектив специалистов по системе "Эльбрус". Школа Святослава Серге-

евича, его мудрые советы и идеи всю жизнь будут для меня руководством к действию.

Я благодарю Г.С. Цейтина, работы которого по модульному программированию оказали большое влияние на подход к разработке программ, описанный в гл. 5.

Выражаю глубокую признательность коллективу Института точной механики и вычислительной техники АН СССР, прежде всего – Г.Г. Рябову, Б.А. Бабаяну, А.Л. Плоткину, В.М. Пентковскому, Т.Ю. Масленниковой, В.Ю. Волконскому, В.М. Гущину, А.Д. Доброву, С.В. Веретенникову, С.В. Семенихину, В.П. Торчигину, Ю.А. Румянцеву, Н.Б. Малышеву, В.Н. Крушнякову, А.Л. Сушенцову, И.П. Колеснику, В.В. Бролю, В.Б. Яковлеву и многим другим. Работы по системе "Эльбрус", выполняемые в ИТМиВТ АН СССР под руководством Б.А. Бабаяна, стали для нас настоящей школой, наше постоянное сотрудничество помогает развитию работ ЛГУ по программному обеспечению МВК "Эльбрус", а высокий научный уровень, результативность и необыкновенный "эльбрусовский дух" этого коллектива являются для нас образцом.

Я признателен также сотрудникам других организаций – пользователей МВК "Эльбрус" и разработчиков его программного обеспечения – Ю.Б. Архангельскому, Н.В. Бардакову, А.Е. Блохину, В.С. Бушуеву, И.С. Голосову, В.П. Горбунову, В.В. Гусеву, Х.Д. Дженибалаеву, О.А. Добрицкому, В.И. Закамскому, В.Б. Зеленскому, Е.А. Зуеву, В.И. Крупянскому, В.С. Лачу, С.П. Макееву, В.А. Маркову, Ю.С. Миленину, Д.Е. Муравьеву, В.Д. Омельченко, Л.Н. Пеньковой, Н.Ю. Петрашковой, И.А. Пиргачу, В.И. Поливанову, А.А. Рейтсакасу, В.А. Сухомлину, В.А. Чаусову, Н.А. Черкашиной, Г.Д. Чинину, А.Т. Шакуро, С.М. Шелестову за сотрудничество, помощь и внимание к нашим работам.

ВВЕДЕНИЕ

В.1. Ориентация на языки высокого уровня

Основная цель данной работы – описание возможностей системы "Эльбрус" для программиста. С этой точки зрения наиболее важная особенность этой системы, положенная в основу ее архитектуры, – ориентация на языки высокого уровня. Рассмотрим ее основные аспекты.

В.1.1. Аппаратная поддержка языковых конструкций. Исторически первым подходом к аппаратной поддержке языков высокого уровня была "фиксация" в аппаратуре какого-либо конкретного языка. Так организованы, например, ЭВМ СИМВОЛ [48], МИР [20]. В системе "Эльбрус" принят другой подход, который, как показала практика, более перспективен и универсален: аппаратно реализованы базовые конструкции и общепринятые механизмы реализации для целого класса распространенных языков программирования (Фортран, Алгол 60, ПЛ/1, Кобол, Паскаль, Алгол-68, Ада, Модула-2 и других). Этот класс языков можно назвать процедурным. Общими для них являются следующие свойства.

1. Процедурный механизм: процедуры с различными классами параметров (параметры-значения, параметры-переменные, параметры-процедуры и др.), рекурсивные процедуры, процедурные типы и объекты (языки: Алгол-68, Клу, Модула-2, Ада).

2. Средства обработки простых значений (целых и вещественных чисел, логических значений, символов) и сложных структур данных: массивов, записей, множеств (шкал), динамических структур и связанных с ними указателей; файлов.

3. Средства структурирования программы: блоки; условные и выбирающие предложения, циклы; исключительные ситуации и операторы их обработки (языки Ада и Клу).

4. Средства модульного программирования: модули, пакеты, абстрактные типы данных (расширение Алгола-68, Ада, Модула-2, Клу).

Все эти языковые элементы присутствуют в Эль-76 и поддерживаются в системе "Эльбрус" аппаратурой и операционной системой.

Процедурный механизм МВК "Эльбрус". Аппаратно реализованы все основные компоненты механизма процедур: вход в процедуру, выход

из процедуры, передача и использование различных классов параметров. Для отображения в аппаратуру языковых понятий "параметр-процедура" и "процедурный объект" введен системный тип данных "метка процедуры". Для исполнения процедур и размещения их локальных данных используются аппаратный стек; для адресации локальных данных введены специализированные *базовые регистры*. Все эти классические механизмы реализации процедур в системе "Эльбрус" существуют уже на аппаратном уровне. Тем самым, имеется единая основа для реализации процедур во всех языках программирования, что снимает ряд рутинных проблем реализации и многие проблемы стыковки модулей на разных языках, весьма остро стоящие для традиционных структур ЭВМ. Однако возможности процедурного механизма МВК "Эльбрус" более широки. Он включает также процедуры с собственным контекстом, элементы которого (собственные локальные данные) могут размещаться в произвольных областях памяти. Это свойство является основой для реализации средств модульного программирования. Кроме того, аппаратура обеспечивает возможность вызова независимой программы как обычной процедуры, что придает программам в системе "Эльбрус" важное для многих языковых приложений свойство динамической расширяемости.

Элементарные типы данных. Совокупность простых типов данных МВК "Эльбрус" покрывает потребности большинства языков высокого уровня. В системе имеются:

– целые числа двух форматов – слово (64 разряда) и полуслово (32 разряда),

– вещественные числа трех форматов – слово, полуслово и удвоенное слово (128 разрядов),

– наборы – обобщение языковых типов данных **bool** (логический), **char** (символьный), **alfa** (короткая строка, размещаемая в слове), **bits** (последовательность битов слова), **bytes** (последовательность байтов слова).

Массивы. В системе "Эльбрус" массивы являются наиболее простыми из структурированных типов данных. Аппаратно реализованы *дескрипторы* (паспорта) одномерных массивов – *векторов* (обычных массивов) и *строк* (упакованных массивов), паспорта многомерных массивов, основные операции над массивами: обращение к элементу массива и изменение его значения. Для векторов и строк имеются аппаратные операции выделения длины и формирования дескриптора подмассива (вырезки). Для реализации упакованных (**packed**) массивов предусмотрена работа с битами, тетрадами и байтами и их смежными группами внутри строки.

Наиболее распространенный способ обработки массивов (поэлементное последовательное обращение в цикле) оптимизируется аппаратурой: параллельно с обработкой текущего элемента выполняется выборка очередных элементов массива. Это в несколько раз сокращает время выполнения многих программ.

Управление памятью. В аппаратуре и ОС реализован гибкий механизм *математической* (виртуальной) памяти, позволяющий системе динамически создавать массивы требуемого для задачи размера, с точностью до слова *физической* (оперативной) памяти. Программисту предоставляется возможность описывать и использовать массивы большого размера (до 2^{20},

т.е. более миллиона, слов на каждый массив) и массивы с динамически изменяемой длиной. Система аппаратными средствами, без потери эффективности, обеспечивает отведение памяти по необходимости. При описании массива память для него не отводится, а формируется специальное слово – заявка. При первом обращении к массиву выделяется математическая память, при обращении к конкретному элементу – физическая. Физическая память отводится только для тех элементов, к которым происходит обращение. Аналогично распределяется память для сегментов объектного кода при исполнении программы, но код размещается сразу в физической памяти. Каждой задаче предоставляется очень большое пространство математической памяти – 2^{32} слов. Система управления памятью МВК "Эльбрус" является удобной базой для динамического распределения памяти в системах программирования, решает проблему размещения больших и динамических массивов.

Управление программой. Набор аппаратных операторов управления МВК "Эльбрус" определяется потребностями языков программирования. Он включает:

– команды переходов (для условных конструкций, операторов перехода и простых циклов),

– команду "переключатель" (для выбирающих предложений),

– команды начала и конца цикла и обращения к параметрам цикла (для циклов с заголовком),

– системный тип данных "метка перехода" и команду динамического перехода по метке (для операторов перехода и структурного перехода),

– команду динамического перехода по ситуации с "продолжением" в виде процедуры ОС (для структурных переходов по динамической ситуации).

Вопросы реализации языков программирования в системе "Эльбрус" рассматриваются в гл. 6.

В.1.2. Динамическая типизация. В традиционных ЭВМ с фон-неймановской архитектурой [48] память имеет линейную однородную структуру, состоящую из последовательности слов, используемых как для хранения программы, так и для размещения данных, причем по содержимому элемента памяти аппаратура во время исполнения программы не имеет возможности определить тип хранимой в нем информации. При исполнении машинных команд данные трактуются в зависимости от семантики команд, а не от типов самих данных. Уровень такой архитектуры ЭВМ значительно ниже, чем уровень языков программирования, имеющих средства описания типов данных. Этот семантический разрыв [48] имеет отрицательные практические последствия для программистов, отлаживающих и использующих программы в такой вычислительной среде. Они в основном выражаются в следующем.

1. После трансляции программы обычно теряется обратная связь между объектным кодом и исходным текстом программы, а также между областями памяти ЭВМ и описанием структур данных в исходном тексте. Это затрудняет отладку программы, для успешного осуществления которой программист вынужден научиться мыслить о своей программе на "машинном" уровне (используя содержимое регистров, распечатку памяти,

кода возврата, номера прерываний и т.п.), а не в привычных терминах используемого языка программирования;

2. Не обеспечивается защита от динамических ошибок, которые не удалось обнаружить при визуальном анализе программы или при трансляции. К наиболее распространенным из них относятся:

– использование неопределенного значения переменной,

– неверная адресация (нарушение защиты памяти) при использовании неконтролируемых указателей (например, в языке ПЛ/1) или адресной арифметики (в ассемблерных языках и в языке Си [11]),

– несоответствие типов данных смыслу операции, например выполнение команды сложения с плавающей точкой над целыми числами (такие ошибки возможны даже в программах на "статических" языках, например, при неправильной передаче параметра внешней процедуре или при несогласованном описании COMMON – блока в разных фортрановских подпрограммах).

Разумеется, большинство этих ошибок можно обнаружить и на традиционных ЭВМ путем применения специальных методов (отладочных трансляторов, интерпретации и т.п.). Однако такой путь приводит, во-первых, к значительной потере эффективности, что неприемлемо для больших программ (например, из-за динамической проверки значения переменной при каждом ее использовании), во-вторых, к несогласованности систем динамической поддержки, построенных для реализации различных языков и, следовательно, к затруднениям при стыковке программных модулей на этих языках, которая часто требуется в больших программах.

Введение тегов. Создание единой эффективно реализованной системы динамической поддержки для различных систем программирования на одном семействе ЭВМ, которая обеспечила бы отладку программы в терминах языка программирования и надежную защиту от динамических ошибок, возможно только путем повышения уровня архитектуры ЭВМ, базисным элементом которой должна быть типизация и структурирование памяти. Именно этот принцип, наиболее важный для реализации языков высокого уровня, положен в основу внутренней организации системы "Эльбрус" и, в конечном счете, на нем основана ориентация системы на языки высокого уровня. В системе "Эльбрус" каждое слово памяти имеет кроме информационной части, содержащей собственно элемент данных, управляющую часть, содержащую *тег* этого элемента – внутренний код его типа. Каждый системный тип данных имеет свой тег, на основе которого аппаратура динамически выполняет выбор нужного варианта операции и контроль типов операндов.

Роль тегов при защите от ошибок. В системе "Эльбрус" на основе тегов осуществляется *контекстная защита* данных и памяти. Рассмотрим системные средства защиты от динамических ошибок трех указанных классов.

1. Неопределенные значения. Специальный тег "пусто" предусмотрен для особого типа данных "пустой объект", представляющего неопределенные значения переменных или элементов массивов. При использовании пустого объекта в операциях, требующих значения некоторого определенного типа (например, в арифметических операциях) возникает прерывание "неверный операнд". Таким образом, использование неопределенного значения переменной обнаруживается немедленно. Важность

такого контроля подтверждается большим практическим опытом переноса на МВК "Эльбрус" пакетов прикладных программ на языках Фортран и Алгол-60 с ЕС ЭВМ и БЭСМ-6 [39, 67]. В пакетах, которые несколько лет эксплуатировались на этих машинах, при первых же тестовых прогонах на МВК "Эльбрус" были обнаружены ошибки, связанные с использованием неопределенных значений переменных. Такой эффект вызван тем, что в системах программирования на традиционных ЭВМ неопределенные значения переменных либо представляются экстремальными числовыми значениями (например, числом с минимальным порядком), либо вообще игнорируются (т.е. начальным значением переменной становится случайное содержимое элемента памяти). Оба эти решения могут привести к ошибкам, которые сложно обнаружить.

2. Защита областей памяти. Адресные типы данных в системе "Эльбрус" имеют свои особые теги. Основным адресным типом данных является дескриптор – аппаратный паспорт вектора. Дескриптор содержит формат массива, его длину, начальный адрес и признаки защиты. Он формируется операционной системой при выделении математической памяти для массива. При обращении к элементу массива, в команде индексации контролируются границы изменения индекса и признаки защиты. Таким образом, программа не может выйти за пределы выделенных для нее массивов, и обеспечивается полная защита памяти между программами. Аналогичным образом обеспечивается защита от выхода за границу *сегмента программы* и *области локальных данных* исполняемой процедуры. Каждая область памяти, в которой размещаются локальные данные процедуры, рассматривается как вектор, дескриптор которого помещается в соответствующий *базовый регистр.* В системе выделена также специальная область памяти, общая для всех задач, в которой размещаются *массивы констант* – массивы, элементы которых постоянны во время выполнения программ. Дескриптор массива констант содержит признак защиты от записи. Попытка модификации элементов такого массива приводит к ситуации "нарушение защиты":

3. Контроль типов. Благодаря аппаратному контролю тегов при выполнении машинных операций использование в них значений недопустимых типов данных полностью исключается: оно приводит к ситуации "неверный операнд". Например, такая ситуация возникает, если операндом операции "сложение" является значение нечислового типа.

Другой пример относится и к контролю типов, и к защите памяти – адресная арифметика. Дескриптор не может быть операндом арифметической операции. Это исключает возможность формирования адреса, указывающего в "чужую" область памяти. Реализация адресной арифметики может быть осуществлена путем размещения адресуемой области памяти в векторе и выполнения арифметических операций над индексами (относительными адресами) в этом векторе. При этом попытка обращения к несуществующему элементу (адресу) приводит к ситуации "граница массива".

Роль тегов для структурирования памяти. В практике программирования часто используются неоднородные структуры данных (массивы из компонент разных типов, таблицы из элементов различной структуры, графы и т.п.). Если информация о представлении такой структуры не хра-

нится во время исполнения программы, то динамически распознать ее и выделить ее компоненты невозможно (в частности, невозможно отличить адрес от другого значения). Между тем проблему динамической декомпозиции неоднородных структур данных требуется решать практически в каждой системе программирования, например при вводе–выводе, сборке мусора, анализе процедурного стека в системе динамической диагностики. В системах программирования для традиционных ЭВМ реализация указанных функций требует хранения в памяти табличной информации о типах данных и структуре программы во время ее исполнения. На МВК "Эльбрус" основой для унифицированного решения этих проблем служат теги. Системная программная компонента (система динамической диагностики, отладчик, процедура обмена) по тегам распознает адресную информацию и восстанавливает полную структуру памяти. Например, если процедуре форматного вывода передан дескриптор массива, содержащего целые, вещественные числа и внутренние дескрипторы, она распознает типы данных по тегам и выводит числовые значения по соответствующим числовым форматам, а внутренние массивы – по тем же правилам поэлементно. Без аппаратных тегов реализация такой процедуры была бы невозможна (точнее говоря, потребовалось бы моделировать теги, передавая процедуре вместе со значениями информацию об их типах).

Теги и динамические языки. Весьма значительную роль играют теги для эффективной реализации *динамических* языков (Лисп, Снобол-4, Рефал и др.), в которых типы переменных не фиксируются во время трансляции и, следовательно, контроль типов должен выполняться динамически. Для таких языков теги являются основой представления информации о типах значений во время исполнения программы. К динамическому классу языков принадлежит Эль-76 [52] – базовый язык высокого уровня в системе "Эльбрус". Динамизм этого языка существенно облегчает разработку системных программ. Подтверждением может служить уже рассмотренный пример процедуры форматного вывода. На статическом языке (например, на языках Паскаль или Алгол-68) ее пришлось бы программировать с использованием средств ослабления контроля типов (вариантных записей, объединенных видов и т.п.), так как в ней требуется обрабатывать структуры данных с компонентами разных типов. Подробно вопросы реализации динамических языков рассмотрены в гл. 6.

Теги и полиморфные операции. Наличие тегов позволило ввести в систему команд МВК "Эльбрус" *полиморфные операции*, применимые к объектам целого класса типов. Полиморфизм – свойство, присущее большинству операций в языках программирования. Например, арифметические операции и операции отношения применимы и к целым, и к вещественным числам; в языке Паскаль символы операций "+", "–" и "*" используются также для обозначения операций над множествами; язык Алгол-68 позволяет доопределить любую операцию для любых типов (например, сложение – для матриц). Это же свойство получило развитие и в языке Ада.

В системе "Эльбрус" свойством полиморфности обладают арифметические операции и операции отношения. При выполнении этих операций операнды приводятся к наиболее общему типу и наибольшему формату. Например, при сложении короткого вещественного и длинного целого числа получается длинное вещественное. Тем самым, аппаратно реализо-

ваны характерные для языков программирования полиморфные свойства арифметических операций, для реализации которых на традиционных ЭВМ компилятор генерирует дополнительные команды приведения.

Другие функции тегов в системе "Эльбрус" носят более специальный характер и рассматриваются в гл. 6.

В.1.3. Эль-76 – базовый язык системы "Эльбрус". Свойства архитектуры, рассмотренные выше, позволили полностью исключить в системе "Эльбрус" ассемблерное программирование и обеспечить *полный перевод программирования на языки высокого уровня*. Для системы "Эльбрус" создано уникальное языковое ядро, через которое программист "видит" аппаратные и программные компоненты системы, – универсальный базовый язык высокого уровня Эль-76. Он обеспечивает сочетание современных средств программирования с доступом ко всем системным возможностям. Язык Эль-76 реализован с максимальной эффективностью – большинство конструкций имеют полную аппаратную поддержку, транслятор выполняет большой набор оптимизаций.

В системе "Эльбрус" язык Эль-76 выполняет следующие функции.

1. Язык системного программирования. На языке Эль-76 запрограммированы все компоненты ОСПО МВК "Эльбрус": ОС, включая систему файлов, трансляторы, система динамической диагностики, комплексатор, отладчик и др. Ни в одной из них не потребовалось программирование в кодах или на языке типа ассемблера. Использование языка высокого уровня резко повысило производительность труда системных программистов, которая на МВК "Эльбрус" достигает нескольких десятков тысяч написанных и отлаженных строк в год [65].

2. Язык проблемного программирования. На языке Эль-76 написан значительный объем проблемных программ: систем управления, систем автоматизации проектирования, пакетов прикладных программ и др. Важными для проблемного программирования являются, кроме общих средств обработки данных и структурирования программ, новые возможности языка Эль-76: средства модульного программирования [10] (для разработки больших программных комплексов) и определяемый синтаксис [54] (для создания командных языков управления проблемными программами).

3. Язык взаимодействия с операционной системой. В МВК "Эльбрус" нет специального языка управления заданиями, его функции выполняет язык Эль-76. Это позволяет записывать задания более наглядно, используя вызовы процедур для обращения к компонентам системы, управляющие конструкции – для анализа хода выполнения задания, условного и повторного выполнения его фрагментов. Через язык Эль-76 доступны и другие средства ОС:

– управление памятью (создание и уничтожение локальных и глобальных массивов, управление массивами переменной длины);

– управление процессами (создание, запуск, синхронизация и уничтожение);

– управление программами (открытие – подготовка к исполнению и запуск).

Другие компоненты ОСПО (текстовый редактор, трансляторы и т.п.) также доступны через язык Эль-76 как стандартные процедуры.

4. Язык управления файлами и базами данных. Средства управления *внешними объектами* (файлами и базами данных) введены в язык Эль-76 как стандартное расширение. Программисту предоставляются три уровня работы с внешними объектами:

– физический обмен (для привилегированных компонент ОС и системы файлов);

– система простых файлов (для системных и прикладных программ);

– система структурированных файлов (для прикладных программ; данный уровень соответствует СУБД и различным методам доступа к файлам).

5. Язык для построения командных языков. Средства расширения языка Эль-76, названные *определяемым синтаксисом* [54], предназначены для определения и использования специализированных командных языков, имеющих простой синтаксис (имена директив, ключевые параметры, умолчания). Синтаксис командного языка задается в виде описаний синтаксических правил, семантика директив – в виде процедур их интерпретации. С помощью этих средств реализованы стандартные расширения Эль-76 – *язык стандартного диалога* и *язык администратора*, а также командный язык Шэл (расширение известного командного языка shell системы UNIX) [49].

Эль-76 – языковая основа совместимости различных моделей МВК "Эльбрус". Эти модели вследствие совершенствования архитектуры комплекса, появления новых моделей различного назначения и производительности не являются совместимыми на уровне системы команд. Однако для всех моделей системы "Эльбрус" обеспечивается программная совместимость на уровне языка Эль-76. Например, программа на Эль-76 может без всяких изменений исполняться и на МВК "Эльбрус-1", и на МВК "Эльбрус-2".

В.1.4. Элементы языковой поддержки в ОСПО. Логическим продолжением принципа аппаратной поддержки процедурного класса языков программирования является создание для МВК "Эльбрус" универсального промежуточного символьно-табличного представления программы [19], отображающего структуру и описания программы в системные понятия МВК "Эльбрус". Любой компилятор в системе "Эльбрус" преобразует программу в *расширенный файл объектного кода* (РФОК), состоящий из собственно *файла объектного кода* (ФОК) и *дополнения к файлу объектного кода* (ДФОК). ФОК содержит *сегменты кода* программы и ее *массивы констант*, ДФОК – *символьно-табличное представление программы*. ДФОК состоит из следующих компонент:

– *справочник локальных словарей* (СЛС) – отображение процедурной структуры программы – содержит имена процедур, их координаты по тексту и коду и ссылки на локальные словари;

– *локальные словари* (ЛС) – отображение описаний на входном языке программирования в машинные понятия МВК "Эльбрус". Например, для каждой локальной переменной ЛС содержит ее идентификатор, формат в стеке, адресную пару (уровень и смещение) и языковый тип. ЛС соответствует области локализации идентификаторов (процедуре, блоку, модулю);

– *справочник соответствия кода тексту* (ССКТ) – таблица соответствия между координатами по коду и по тексту программы.

РФОК является в системе "Эльбрус" универсальным интерфейсом между компиляторами и общими системными программными компонентами языковой поддержки. Каждый компилятор должен формировать стандартную структуру РФОК. При этом общие системные средства обеспечивают:

– открытие и запуск программы по ее ФОК (*операционная система*);

– диагностику динамической ошибки в терминах программы на исходном языке с указанием номера строки, фрагмента исходного текста, имен и текстовых координат вызванных процедур, имен и значений их локальных переменных (*система динамической диагностики*);

– сборку программ из процедур, написанных на одном или нескольких языках программирования (*комплексатор*);

– отладку программы в терминах исходного текста (*отладчик*);

– получение словаря идентификаторов программы – документа, содержащего информацию о точках и характере использования идентификатора (*программа распечатки словаря идентификаторов*). Для обеспечения этой функции системы от компилятора требуется генерация еще одной стандартной структуры – *файла ссылок* на вхождения идентификаторов.

Все перечисленные программные компоненты ОСПО используют РФОК. Например, система динамической диагностики, вызываемая при прерывании, используя координаты места ошибки в объектном коде, информацию из ДФОК и теги слов в памяти, анализирует структуру процедурного стека и выдает его распечатку в терминах исходного текста программы. Средства языковой поддержки ОСПО МВК "Эльбрус" можно рассматривать как часть любой системы программирования, так как они "понимают" любой язык, поскольку его компилятор соблюдает стандартную структуру РФОК. Поэтому в системе "Эльбрус" можно говорить о единой многоязыковой системе программирования, объединяющей системы программирования на базе различных языков и базовые средства языковой поддержки.

В.2. Система программирования МВК "Эльбрус"

В.2.1. Структура и компоненты. Система программирования МВК "Эльбрус" в ее нынешнем виде создана менее чем за десять лет специалистами Москвы (ИТМиВТ АН СССР), Новосибирска (НФ ИТМиВТ), Ленинграда (ЛГУ), Таллина (ИК АН ЭССР), Ростова-на-Дону (РГУ) и Кишинева (ИМ с ВЦ АН МССР). Эта система является уникальной как по общему подходу и принципам разработки, так и по набору входящих в нее языков и компонент.

Система программирования МВК "Эльбрус" состоит из следующих частей.

1. Базовые инструментальные и технологические средства:

– язык и система программирования Эль-76,

– система динамической диагностики,

– многоязыковый комплексатор,

– отладчик программ в терминах исходного текста,

– система получения словаря идентификаторов программы,

– технологические пакеты для разработки трансляторов [8]: ИНТЕРТЕКСТ – пакет для работы с текстовой формой программы, ИНТЕРФОК –

НПО ... Орел
На, ...

пакет для работы с РФОК, ИНТЕРКОД – пакет для генерации и оптимизации кода конкретной модели МВК, ИНТЕРПАМ – пакет для динамической работы с памятью задачи.

2. Производственные системы программирования на базе распространенных языков – Фортран, Алгол-60, Кобол, ПЛ/1, Алгол-68, Симула-67, Паскаль.

3. Специализированные и экспериментальные системы программирования:

– Клу (для разработки больших программных комплексов с использованием абстрактных типов данных),

– Лисп, Рефал, Снобол-4, Плэнер (для решения задач обработки нечисловой информации),

– АБВ (для разработки систем программирования),

– Сол (для решения задач моделирования),

– МИС (для автоматизации разработки пакетов прикладных программ).

4. Диалоговые системы программирования:

– Форт (система программирования на базе расширяемого языка FORTH-83,

– ДИАШАГ (система для разработки и отладки программ в интерактивном режиме на основе пошагового транслятора с языка Алгол-60).

Система программирования МВК "Эльбрус" проектировалась и создавалась как открытая система. Работы по ее развитию продолжаются: создаются системы программирования для популярных современных языков Си, Ада, Модула-2 и Пролог; разрабатываются системные средства управления программными конфигурациями и версиями [53] – базовая программная поддержка технологии программирования.

В.2.2. Принципы разработки. В отличие от систем программирования для наиболее распространенных ЭВМ (БЭСМ-6, ЕС ЭВМ, СМ ЭВМ), структура которых формировалась постепенно, десятилетиями, разными коллективами и поэтому не имеет единой концептуальной основы, система программирования МВК "Эльбрус" с самого начала создавалась по определенным принципам, отражающим замысел и возможности архитектуры, преемственность программного обеспечения и потребности его развития.

Последующее расширение возможностей системы в духе ее первоначального замысла не привело к необходимости серьезных изменений в программном обеспечении.

Преемственность программ обеспечивается двумя свойствами, которыми обладает большинство систем программирования для МВК "Эльбрус":

– во входных языках предусмотрены диалекты для распространенных ЭВМ. Настройка на диалект выполняется с помощью управляющих карт (прагматов). Например, для языков, получивших распространение на ЕС ЭВМ (Фортран, Кобол, ПЛ/1, Алгол-60), обеспечивается совместимость с ЕС ЭВМ "на уровне колоды перфокарт";

– в любой программе могут использоваться библиотеки (пакеты) программ на том же языке или на других языках.

Унификация. Ее основой является единая аппаратная поддержка процедур, базовых структур данных и единая структура РФОК (В.1.4). Цель

унификации – облегчение работы программистов и обеспечение программирования на разных языках в рамках одной программы. В системе программирования МВК "Эльбрус" унифицированы:

– способы реализации основных языковых конструкций,

– структура РФОК (стандартная форма представления программы и ее семантики),

– форма обращения к трансляторам, порядок и смысл их параметров,

– форма управляющих карт, задающих основные режимы работы транслятора.

Единая форма обращения к трансляторам обеспечивает применение одной и той же директивы для запуска любого транслятора в диалоге. Единая форма прагматов введена для удобства пользователей (впрочем, при наличии традиционных для конкретного языка форм прагматов последние также реализованы – например, управляющие комментарии в языке Паскаль).

Для генерации РФОК и объектного кода на конкретной модели МВК "Эльбрус" в системе имеются технологические пакеты ИНТЕРФОК и ИНТЕРКОД. Их использование разработчиками системы программирования автоматически обеспечивает включение компилятора в общую многоязыковую систему программирования (стыковку с общими средствами языковой поддержки) и перенос транслятора с одной модели МВК на другую, например, с МВК "Эльбрус-1" на МВК "Эльбрус-2".

Полнота использования возможностей МВК "Эльбрус". Для разработчика транслятора система "Эльбрус" автоматизирует значительную часть рутинной работы: реализацию процедур, массивов, основных операторов, генерацию РФОК и т.п. Фактически в распоряжение системного программиста представлена базовая система построения трансляторов. Это позволяет сосредоточиться на более творческих проблемах адекватного отображения особенностей языка программирования в аппаратные средства МВК "Эльбрус". В процессе разработки трансляторов глубокому анализу подверглись все аспекты архитектуры МВК "Эльбрус", был использован полный спектр ее возможностей, как базовых (процедурный механизм, стек), так и более сложных (технические метки, аппаратная косвенная адресация, процедуры с собственным контекстом). Подробное изложение этих вопросов, представляющее интерес для системных программистов, дано в гл. 6.

Другая сторона этой проблемы важна для программистов-пользователей. В большинстве входных языков реализованы расширения, обеспечивающие более полное использование возможностей аппаратуры и ОС: различных форматов и простых типов данных, интерфейса с ОС, средств работы с внешними объектами.

Полнота и новизна версий входных языков. Например, в системе программирования Фортран – Эльбрус реализовано расширение языка Фортран-77, в системе программирования Паскаль – Эльбрус – расширение международного стандарта языка Паскаль [81], принятого в 1983 г. Во входных языках системы программирования МВК "Эльбрус" сняты многие традиционные ограничения, например разрешены рекурсивные процедуры в стандартном диалекте Фортрана.

Исследования по созданию экспериментальных систем программирования. Архитектура МВК "Эльбрус" позволила эффективно реализовать

новые черты языков программирования, а ряд уже известных и реализованных за рубежом языков осуществить новыми, более эффективными и современными методами. Как уже подчеркивалось, возможности системы "Эльбрус" стимулируют перспективные исследования в области создания трансляторов. Например, для системы "Эльбрус" была разработана первая отечественная реализация языка Клу [83], основанного на концепции абстрактных типов данных, впервые в практике отечественного программирования, новыми методами реализован динамический язык символьной обработки Снобол-4 [23], создан компилятор с языка системного программирования АБВ [62], имеющего нетрадиционные динамические свойства. В направлении исследования и более полного использования возможностей аппаратуры МВК "Эльбрус" развивается базовая система программирования Эль-76. Наиболее существенное ее новшество – средства модульного программирования [10]. В язык Эль-76 введены определяемые типы данных. Разработана модель новой редакции языка Эль-76, получившая название ДАТД (динамическое управление абстрактными типами данных) [10].

В.2.3. Обзор работ по системе программирования МВК "Эльбрус". В этой части введения рассмотрим наиболее важные результаты работ по отдельным системам программирования для МВК "Эльбрус", описание которых не вошло в основную часть книги.

Система программирования Эль-76 [52] – базовая система программирования МВК "Эльбрус". Особое значение этой системы программирования состоит не только в универсальности языка Эль-76 (В.1.3). Эта система является эталоном качества реализации языков программирования для МВК "Эльбрус" и школой для многих разработчиков трансляторов. Ее авторы В.М. Пентковский, В.Ю. Волконский, Ю.А. Румянцев, Н.Б. Малышев, Е.Н. Чернис и другие положили начало всему комплексу работ по системам программирования для МВК "Эльбрус". Транслятор языка Эль-76 обеспечивает высокую скорость трансляции и эффективность объектного кода, выполняет смешанные вычисления и оптимизации (например, оптимизации обращения к элементам массивов в циклах, частичное выполнение предложений с постоянными компонентами, оптимизации управляющих конструкций и др.).

Технологические пакеты (ИНТЕР-пакеты) [8]. Методы генерации и оптимизации кода, используемые в системе Эль-76, были положены в основу технологических пакетов для разработки трансляторов ИНТЕРФОК и ИНТЕРКОД, ведущими разработчиками которых являются В.Ю. Волконский, В.Н. Крушняков и А.Л. Сушенцов. Пакет ИНТЕРФОК позволяет разработчику транслятора абстрагироваться от конкретного представления расширенного файла объектного кода, генерировать и анализировать его в терминах его древовидной структуры и исходного текста программы. Пакет ИНТЕРКОД генерирует код конкретной модели МВК "Эльбрус" и выполняет некоторые его оптимизации. Пакет обеспечивает для создателей трансляторов работу в терминах базовой (виртуальной) системы команд МВК "Эльбрус", скрывающей различия между моделями системы.

Система программирования Фортран [5] – исторически первая система программирования на МВК "Эльбрус" (наряду с базовой системой Эль-76).

Разработчики первой версии фортран-транслятора (Н.Ю.Петрашкова, Н.А.Черкашина, С.М.Шелестов) сыграли огромную роль в освоении и развитии средств языковой поддержки системы "Эльбрус". К настоящему времени система программирования Фортран прошла значительный путь развития. Она существует в двух вариантах. В первом реализован стандартный Фортран (ГОСТ 23056-78) и некоторые диалекты с элементами входных языков трансляторов Фортран-Дубна и Фортран-IV ОС ЕС ЭВМ. Второй, более поздний вариант представляет собой систему программирования с оптимизирующим компилятором, входной язык которого является расширением последней версии языка – Фортран-77. Оба варианта в совокупности представляют собой уникальную фортран-систему программирования, которая по эксплуатационным характеристикам приближается к эталонной системе программирования Эль-76, а по качеству и полноте реализации (по отзывам специалистов) превосходит все эксплуатируемые фортран-трансляторы ЕС ЭВМ. Объем программного обеспечения на языке Фортран, перенесенного на МВК "Эльбрус" с ЕС ЭВМ и БЭСМ-6, составляет более 500 тысяч строк. Оно включает, в частности, известные зарубежные пакеты FORPACK, LINPACK, EISPACK [39]; пакеты прикладных программ ПРОЗА – по расчету оболочек вращения и "Статистика" – по статистической обработке данных, разработанные в ЛГУ. Система программирования Фортран взаимодействует со всеми средствами языковой поддержки ОСПО, обеспечивает использование процедур и пакетов программ на других языках программирования.

Система программирования Алгол-60 [50] принадлежит также к числу первых систем программирования на МВК "Эльбрус". Разработчики ее первого действующего варианта (Л.Н.Пенькова и Э.А.Семенова), как и создатели первой версии фортран-транслятора, являются пионерами освоения и развития системы программирования МВК "Эльбрус". Первая версия системы программирования Алгол-60, как и система Фортран, доведена до высоких эксплуатационных показателей, состыкована со средствами языковой поддержки ОСПО, выполняет ряд оптимизаций объектного кода. Входным языком этой версии системы является полный язык Алгол-60, в соответствии с модифицированным сообщением [4], расширенный процедурами ввода-вывода языка Алгамс (не реализованы лишь числовые метки и динамические собственные массивы). Второй вариант системы программирования Алгол-60 – система Алгол-Эльбрус [50] – был разработан с двоякой целью:

– расширение входного языка диалектами распространенных систем программирования БЭСМ-Алгол, Алгол-ГДР, Алгол-ЕС ЭВМ и средствами обращения к интерфейсу ОС "Эльбрус",

– отработка методики использования технологических пакетов ИНТЕРТЕКСТ, ИНТЕРФОК и ИНТЕРКОД [8] для создания трансляторов (на системе Алгол-Эльбрус эти пакеты проходили опытную эксплуатацию).

Входной язык системы Алгол-Эльбрус объединяет все указанные диалекты и имеет русский вариант нотации, что (наряду со средствами обращения к ОС) сближает его по форме с языком Эль-76.

Систему программирования Алгол дополняет разработанная в ЛГУ диалоговая система программирования ДИАШАГ [59], включающая пошаго-

вый транслятор с языка Алгол-60 и предназначенная для разработки и отладки программ в диалоговом режиме. Входной язык системы ДИАШАГ согласован с входным языком первой версии системы программирования Алгол. Система ДИАШАГ рассмотрена в п. 4.4.

Система программирования Алгол-68 [13] является полной реализацией русского варианта языка Алгол-68 [55], расширенного средствами модульного программирования [47]. Это первая отечественная реализация Алгола-68 со столь мощным входным языком. Ведущие разработчики системы – В.В.Броль, В.М.Гущин и В.Б.Яковлев. Реализация Алгола-68 ставит перед разработчиками транслятора много сложных проблем, связанных с описаниями видов и операций, статическим видовым контролем, приведениями, разнообразием форм синтаксиса конструкций, динамической поддержкой [3]. Тем более ценно, что система программирования "Алгол-68-Эльбрус" не содержит каких-либо ограничений, характерных для других реализаций. Например, полностью реализован контроль области действия. Многопросмотровая схема транслятора позволила включить в него оптимизации кода и обеспечить высокое качество диагностики ошибок. Следует также отметить организацию обмена. Реализован весь сложный аппарат обмена, предусмотренный разработчиками языка (книги, каналы, файлы, процедуры доступа, форматный обмен). С помощью системы Алгол-68-Эльбрус осуществлен перенос с ЕС ЭВМ двух разработанных в ЛГУ пакетов прикладных программ: по решению задач механики деформируемого твердого тела и по решению уравнений Навье – Стокса.

Система программирования Симула-67 [29] представляет собой реализацию популярного языка программирования Симула-67 [27], построенного на базе языка Алгол-60 и предназначенного для разработки больших программ и моделирования систем с дискретными событиями. При исполнении программы на языке Симула-67 возникает не последовательность вызванных блоков и процедур, как в традиционных языках, а дерево квазипараллельных процессов. Между его элементами возможны сопрограммные связи. Основное средство структурирования программы и данных – концепция класса (агрегата данных и процедур), которая близка к понятию абстрактного типа данных и отличается от него лишь отсутствием инкапсуляции. При реализации были использованы средства ОС для управления сопроцессами, аппаратный вызов процедур с собственным контекстом, команды поиска по списку. Однако авторы реализации отмечают, что отличие механизма исполнения программы в Симуле-67 от традиционного накладывает дополнительные требования на систему динамической диагностики, возможность использования в программе большого числа объектов-сопроцессов (до нескольких десятков тысяч) – на операционную систему, а сложные программные структуры (классы и их сочленения) – на структуру РФОК. Это является стимулом к дальнейшему развитию средств языковой поддержки ОСПО МВК "Эльбрус".

Выделим следующие особенности реализации в системе Симула-Эльбрус:

– тщательность и полнота реализации системных классов SIMSET (для работы со списками) и SIMULATION (для задач моделирования);

– средства трассировки при исполнении программы (эта возможность весьма важна для отладки программ со сложными связями между объектами);

– технология разработки транслятора. Его анализирующая часть получена автоматически, путем преобразования описания синтаксиса в синтаксические таблицы и затем – в анализатор, работающий методом рекурсивного спуска.

Авторы системы – Х.Д.Дженибалаев, Т.А.Жердер, В.В. Зеленский, И.В.Титрова, В.Т.Фоменко.

Система программирования Лисп [57]. Входной язык системы является расширением языка INTERLISP [85] – одной из последних и наиболее развитых версий широко распространенного языка Лисп для обработки списочной и символьной информации. Язык расширен средствами работы с дополнительными структурами данных, отсутствующими в языке Лисп – массивами, строками, файлами. Обеспечивается совместимость с известной системой программирования Лисп-БЭСМ-6 [46]: реализованы все ее стандартные функции. В системе Лисп-Эльбрус использованы оригинальные методы реализации, основанные на возможностях МВК "Эльбрус": система динамического распределения списочной памяти объемом до 2^{29} лисповских ячеек, основанная на системе управления памятью ОС "Эльбрус"; реализация обращения к переменным без поиска по ассоциативному списку; параллельная сборка мусора. Автор системы – А.А.Рейтсакас.

Интеллектуальная система программирования МИС [77] представляет собой адаптированную для МВК "Эльбрус" версию широко известной системы программирования ПРИЗ со входным языком Утопист [72]. Система ПРИЗ – одна из первых в мире систем с автоматическим синтезом программ. Ее основное назначение – автоматизация разработки пакетов прикладных программ для решения инженерных задач, однако фактически она применяется в гораздо более широкой области, включающей задачи искусственного интеллекта и разработку систем управления базами данных и знаний. Система МИС – первая интеллектуальная система программирования для МВК "Эльбрус". Она в основном запрограммирована на языке Фортран, ряд системных модулей – на языке Эль-76. При переносе системы на МВК "Эльбрус" использовался многоязыковый комплексатор. Основные разработчики системы – М.И. Кахро, А.П. Калья, М.Б. Мацкин, Л.Л. Томберг, А.Л. Шмундак, руководитель разработки – Э.Х. Тыугу.

Инструментальный комплекс "Темп" [26] и интерпретатор АК ЛГУ [73]. Одним из главных принципов проекта "Эльбрус" является принцип параллельной разработки аппаратуры и программного обеспечения. Как следствие, большинство описанных разработок по системе программирования, а также многие работы по пакетам прикладных программ и другим программным системам для МВК "Эльбрус" полностью или частично выполнялись при отсутствии МВК "Эльбрус" у организации-разработчика. При этом неоценимую помощь оказали инструментальный комплекс "Темп" для БЭСМ-6 и интерпретатор АК ЛГУ для ЕС ЭВМ, предназначенные для разработки и отладки программ на языке Эль-76 с использованием распространенных ЭВМ.

Инструментальный комплекс "Темп" создан в НФ ИТМ и ВТ. Он содержит монитор, текстовый редактор, транслятор с языка Эль-76 в код МВК "Эльбрус-1" и ряд сервисных компонент (документатор, редактор ФОК, структуратор программ на языке Эль-76, библиотеку стандартных функ-

ций). Комплекс "Темп" начиная с 1979 г. был использован для разработки и отладки почти всех систем программирования, о которых идет речь в этой книге. Новой чертой комплекса "Темп" является включение в него "большого" интерпретатора системы команд МВК "Эльбрус-2", включая привилегированные операции. Руководитель проекта – В.А. Марков.

В 1981 г. в ЛГУ под руководством Н.Ф. Фоминых был создан интерпретатор АК ЛГУ, который сыграл важную роль при разработке и отладке пакетов прикладных программ и других проблемных программных комплексов на языке Эль-76. Интерпретатор построен по иному принципу, нежели комплекс "Темп": он не моделирует систему команд МВК "Эльбрус", а является интерпретатором самого языка Эль-76 (для интерпретации используется свой внутренний язык). Кроме того, он моделирует некоторые операции системы простых файлов МВК "Эльбрус" (комплекс "Темп" предоставляет собственные, более простые, средства ввода-вывода). Такого набора возможностей оказалось достаточно для отладки пакетов прикладных программ и интерпретирующих систем программирования. Например, на интерпретаторе АК ЛГУ велась отладка системы Лисп-Эльбрус.

Система программирования МВК "Эльбрус" постоянно развивается. Разработчики системы ведут активную деятельность по ее совершенствованию, сопровождению, пропаганде и внедрению.

ГЛАВА 1

МЕТОДЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ЭЛЬ-76

Данная глава представляет собой практическое руководство по программированию на языке Эль-76 [52] – базовом языке высокого уровня системы "Эльбрус". Материал главы является логическим продолжением и развитием учебника [60]. В ней отражены дальнейшая эволюция языка, опыт его практического использования и преподавания в Ленинградском университете. В главе описаны основные понятия и конструкции языка Эль-76, его функции и методы применения в системе "Эльбрус". Рассмотрены ядро языка, средства работы с внешними объектами, средства модульного программирования, определяемый синтаксис, средства взаимодействия с операционной системой. Описано также стандартное расширение языка Эль-76 – командный язык диалога. Изложение построено на сравнении с другими, более известными языками программирования и на содержательных примерах.

Содержание главы можно рассматривать и как практическое введение в программирование в системе "Эльбрус", ориентированное на программистов, знакомых с одним или несколькими языками программирования. Фактически при описании элементов языка Эль-76 рассматриваются и соответствующие элементы самой системы "Эльбрус", так как язык Эль-76 задуман и реализован как языковое ядро, через которое программисту доступны все возможности системы.

Изучение языка Эль-76 и основ системы "Эльбрус" представляет несомненный интерес не только для программистов, по роду своей деятельности связанных с этой системой, но и для более широкого круга специалистов, студентов и аспирантов, желающих изучить современные идеи и методы программирования. Для более углубленного изучения языка Эль-76 рекомендуется его авторское описание [52].

В данной главе описано наиболее известное, широко применяемое пользователями системы "Эльбрус" подмножество языка Эль-76, основанное на динамическом управлении типами. В нем типы объектов, обрабатываемых программой, не фиксируются в описаниях переменных, а контролируются при исполнении программы. Номенклатура типов ограничена встроенными в язык (стандартными) и аппаратно распознаваемыми типами. Во втором издании монографии [52] дано описание расширенного языка

Эль-76, включающего определяемые типы: диапазоны, массивы, структуры, процедурные и модульные типы, средства организации списковых структур, а также средства управления степенью динамизма контроля типов. Таким образом, на расширенном языке Эль-76 можно программировать и в "динамическом", и в "статическом" стиле либо использовать их сочетание.

1.1. Пример программы

Изложение основных понятий языка Эль-76 начнем с простого примера программы, использованного (в несколько иной форме) в работах [60] и [52]. Программа вычисляет скалярное произведение двух векторов из 100 чисел. Значения их элементов вводятся из входного файла, результат выводится в выходной файл. В программе описана функция, вычисляющая скалярное произведение векторов.

```
Пример1 = проп (ввод, вывод)
начало % описания векторов а, в и процедуры скал
  конст а = лок вект [100] ф64,
        в = лок вект [100] ф64;
  процедура скал = функция (x, y)
      (ф64 c := 0;
      для k до длина x − 1
      цикл
          c := * + x [k] * y [k]
      повторить;
      c); % значение c − результат функции скал
% основная часть программы
 читф (ввод, а, в);
 запф (вывод, скал (а, в))
конец
```

Для сопоставления особенностей языка Эль-76 со свойствами более привычных языков программирования приведем программу на Алголе-68, наиболее близкую к данной как по форме написания, так и по семантике.

```
begin со описания массивов a, b и процедуры скал со
   [0 : 99] real a, b;
proc скал = (ref [ ] real x, y) real:
(real c := 0;
for k from 0 to upb x do
   c + := x[k] * y[k]
od;
c); со значение c − результат функции скал со
со основная часть программы со
read ((a, b));
print (скал (a, b))
end
```

Рассмотрим основные черты сходства и отличия этих программ.

Сходство. Программы сходны между собой по общей структуре, назначению и форме записи отдельных компонент. Обе программы состоят

из описаний векторов *а* и *в*, описания процедуры-функции *скал* и основной части – операторов вызова стандартных процедур ввода-вывода. Схожи между собой и внутренние конструкции процедуры *скал*:

– описание переменной *с* с начальным значением 0;

– цикл с параметром *k*; параметр цикла считается описанным неявно своим вхождением после слова **for** (для);

– использование переменной *с* перед закрывающей скобкой тела процедуры; в обоих языках оно означает, что результатом *замкнутого предложения* и процедуры *скал* является значение *с*.

Как и в Алголе-68, в языке Эль-76 сложные управляющие конструкции (*закрытые предложения*) имеют "открывающую и закрывающую скобки" в виде служебных слов.

Имеется сходство и в элементарных конструкциях. Одинаково записываются присваивание, арифметические операции, обращение к элементу массива. Конструкция ":= * +" языка Эль-76 аналогична конструкции "+ :=" Алгола-68 и означает "увеличить значение на".

Нумерация элементов векторов в обеих программах одинакова (однако для этого в программе на Алголе-68 пришлось явно указать обе границы). В системе "Эльбрус" и в языке Эль-76 элементы вектора нумеруются начиная от нуля.

Отличия. Основные отличия программ связаны с различной трактовкой данных и описаний.

В программе на Алголе-68 в каждом описании указан вид (тип) каждого идентификатора: переменной, константы, параметра. Например, для переменной *с* зафиксирован вид **real**, для параметра *x* – вид **ref [ ] real** (имя вектора из элементов вида **real**). В описании процедуры *скал* зафиксирован вид результата – **real**. Наконец, неявно зафиксирован и вид самой процедуры *скал* – **proc (ref [ ] real, ref [ ] real) real**, т.е. процедура с двумя параметрами-векторами из вещественных элементов и с вещественным результатом. Аналогичным свойством обладают языки Паскаль, Ада, Модула-2 и многие другие. Все они относятся к классу *статических* языков, в которых тип любого объекта должен быть либо задан в тексте программы, либо ясен из ее контекста, а контроль типов (проверка правильности использования объектов в операциях) выполняется при трансляции.

В отличие от Алгола-68 в языке Эль-76 имеются динамические типы, при использовании которых в программе в описании фиксируется не тип переменной, а лишь ее *формат* – размер значения в памяти. Переменная *с* имеет формат **ф64** (ее значение занимает слово – 64 разряда). Такой же формат имеют элементы векторов *а* и *в* (он задан в их описании). По умолчанию параметры *x* и *y* также имеют формат **ф64**, фактическими параметрами являются указатели векторов, каждый из которых (как самостоятельный объект) занимает слово. В описании функции *скал* задано лишь служебное слово **функция** – признак наличия результата.

Таким образом, язык Эль-76 допускает значительную свободу в обращении с данными, что, как будет видно из дальнейшего, весьма важно для разработки системных программ. Например, в выражении $x[i] * y[i]$ типы операндов транслятору не известны, и контроль типов выполняется динамически. Если процедуре *скал* ошибочно будет передано вместо указателя вектора число или если в результате ввода (из-за ошибки во входных дан-

ных) значение какого-либо элемента вектора *а* или *в* останется неопределенным, то в операции индексации (умножения) будет зафиксирована динамическая ошибка. При использовании определяемых типов в программе на языке Эль-76 указанное различие программ можно устранить.

В отличие от Алгола-68 в языке Эль-76 принята русская лексика. Впрочем, в русском варианте Алгола-68 служебные слова отличаются от служебных слов Эль-76 лишь незначительно.

Служебные слова языка Эль-76 при наборе или набивке изображаются с предшествующим символом "подчерк" (_). За служебным словом либо должен идти пробел, либо может следовать очередной символ исходного текста, отличный от буквы или цифры. В отличие от Алгола-68 пробелы внутри идентификаторов не допускаются.

На менее существенных отличиях двух программ (в форме записи операций обмена, способах указания файлов для ввода-вывода) здесь останавливаться не будем.

Выводы. Проанализируем приведенные программы с точки зрения их возможностей и эффективности. Как уже отмечалось во введении, основные операции системы "Эльбрус" (например, арифметические) обладают *полиморфизмом*, т.е. применимы к целому классу типов. Соответствующий класс объектов для арифметических операций в системе "Эльбрус" называется *арифметическим классом*. Эти свойства аппаратуры наследуются и языком Эль-76, поскольку каждой его базовой операции соответствует машинная команда. Следовательно, программа вычисления скалярного произведения применима как для целых, так и для вещественных чисел – элементов векторов *а* и *в* (типы их значений определяются при вводе). При этом полиморфизм не приводит к потере эффективности, так как в арифметических операциях аппаратный контроль типов выполняется параллельно с разбором различных сочетаний типов операндов и исполнением соответствующих вариантов операции.

Программа на Алголе-68, ввиду статического контроля типов, может работать только с вещественными числами. Даже если входные данные – целые числа, то при вводе они будут преобразованы в вещественные, что приведет к потере эффективности, поскольку операции с плавающей точкой выполняются примерно в полтора раза дольше. Программу на Алголе-68 можно переписать так, чтобы она могла работать как с целыми, так и с вещественными числами. Но для этого фактически придется прибегнуть к моделированию тегов: описать объединенный вид "целое или вещественное" и доопределить операцию умножения над значениями этого вида, явно выписав все возможные варианты [52]. Модифицированная версия программы на Алголе-68 будет работать еще медленне, так как элементы динамического контроля запрограммированы явно.

Даже из такого простого примера видно, насколько большие возможности дает динамическая типизация. Она обеспечивает полиморфность операций, гибкость в работе с данными и является разумным компромиссом между противоречивыми требованиями надежности и эффективности. При отсутствии этого основного свойства в вычислительной системе ее базовый язык высокого уровня, по-видимому, проектировался бы либо как статический – для повышения надежности (как в системе Барроуз [48]), либо как бестиповый – для обеспечения полиморфности (как язык

ЯРМО для БЭСМ-6 [22]). В обоих случаях какое-либо из трех важнейших свойств базового языка – надежность, полиморфность или эффективность – было бы потеряно. Язык Эль-76 объединяет в себе все эти свойства.

1.2. Классификация и обзор типов данных

Мощность и удобство языка программирования в значительной степени определяются набором его типов данных и операций над ними. Поэтому подробное описание языка Эль-76 будет вестись "от данных": прежде всего рассматривается классификация типов данных, затем – каждый класс типов в отдельности: изображение (внутреннее представление) объектов, операции, особенности.

Совокупность типов данных языка Эль-76 играет в системе "Эльбрус" двоякую роль. С одной стороны, она обеспечивает использование в базовом языке возможностей аппаратуры (в ее нынешнем виде): большинство типов данных языка Эль-76 имеет прямую аппаратную поддержку и свои особые теги в системе. С другой стороны, она является базой для мобильности программ: при дальнейшем развитии системы "Эльбрус" архитектура ее новых моделей может отличаться от существующей, но типы данных Эль-76, их представление и операции над ними будут сохранены и реализованы (хотя, возможно, и не полностью аппаратно).

В табл. 1 приведена полная классификация простых типов данных языка Эль-76. Даны названия типов, названия объединяющих их классов и служебные слова, обозначающие их теги (если такие слова имеются в языке). Эти служебные слова (например, **деск**) обозначают соответствующие константы типа "целое 64".

В языке Эль-76 имеются два формата *целых* (32 и 64 разряда) и три – *вещественных чисел* (32, 64 и 128 разрядов). *Набор* – это обобщение по-

Таблица 1. **Классификация простых типов данных Эль-76**

<table>
<tr><td>целое 32 (цел32)</td><td rowspan="2">класс целых</td><td rowspan="3">обобщенный класс целых</td><td rowspan="6">арифметический класс</td></tr>
<tr><td>целое 64 (цел64)</td></tr>
<tr><td>набор (наб)</td><td></td></tr>
<tr><td>вещественное 32 (вещ32)</td><td rowspan="3" colspan="2">класс вещественных</td></tr>
<tr><td>вещественное 64 (вещ64)</td></tr>
<tr><td>вещественное 128 (вещ128)</td></tr>
<tr><td>пусто 32 (п32)</td><td rowspan="2">пустые объекты</td><td></td><td></td></tr>
<tr><td>пусто 64 (п64)</td><td></td><td></td></tr>
<tr><td>указатель вектора (деск)</td><td rowspan="7" colspan="2">указатели</td><td rowspan="10">класс адресной информации</td></tr>
<tr><td>указатель паспорта массива</td></tr>
<tr><td>указатель динамической ситуации</td></tr>
<tr><td>указатель оперативного объекта связи</td></tr>
<tr><td>указатель модульного объекта</td></tr>
<tr><td>указатель процесса</td></tr>
<tr><td>указатель задачи</td></tr>
<tr><td>метка процедуры (мпроц)</td><td rowspan="3" colspan="2">метки</td></tr>
<tr><td>метка перехода (мпход)</td></tr>
<tr><td>семафор</td></tr>
</table>

нятия машинного слова, допускающее наряду с разбивкой на биты разбивку по тетрадам и байтам и более богатую систему базовых операций.

Пустые объекты форматов **ф32** и **ф62** введены для представления неопределенных значений переменных. Сами эти объекты обозначаются, соответственно, **пусто32** и **пусто64**. Использование пустого объекта в операции, требующей какого-либо определенного типа объекта (например, в арифметической) считается динамической ошибкой и приводит к прерыванию.

Указатели служат для обозначения *составных объектов* – векторов, многомерных массивов, модульных объектов, оперативных объектов связи с внешними объектами (файлами, контейнерами и справочниками), объектов, предназначенных для управления вычислениями (динамических ситуаций, процессов, задач). Указатель, как правило, формируется операционной системой в результате исполнения *генератора объекта* заданного класса. Таким образом, любой составной объект из перечисленных выше можно создать в любом месте программы с помощью генератора. Такой подход обеспечивает гибкость работы со сложными структурами данных и соответствует общему принципу динамизма в системе "Эльбрус".

Метки – это аппаратные типы данных для представления информации о программе, ее фрагментах и точках во время исполнения. *Метка процедуры* соответствует языковому понятию процедуры и содержит информацию, необходимую для ее вызова (ссылки на *код* и *контекст* процедуры). *Метка перехода* соответствует языковому понятию метки и помечает выделенную точку в программе, на которую может быть выполнен переход.

Семафор – аппаратный тип данных, соответствующий классическому понятию двоичного семафора, введенному Э. Дейкстрой. Семафоры – базовое средство синхронизации параллельных процессов.

1.3. Общие операции

В языке Эль-76 ряд основных операций можно выполнять над простыми объектами любого типа. Однако семантика и даже сама их совокупность отличаются от традиционной.

Прежде всего, простой объект можно присваивать переменной или элементу массива и передавать как параметр в процедуру. При присваивании требуется, чтобы объект помещался в соответствующую область памяти. При передаче параметра формат фактического параметра должен совпадать с форматом в описании формального параметра или быть меньше (как правило, оба по умолчанию равны **ф64**).

1.3.1. Выделение и проверка типа. Данная группа операций отражает особенности Эль-76 как динамического языка. Помимо неявного аппаратного контроля типов программист может использовать и явные операции проверки типа (тега) и формата. Такие операции особенно важны для системных программ: при написании полиморфных процедур (например, форматного вывода) и при реализации интерпретационных проверок в системах программирования. Динамический тип текущего значения объекта x определяется операцией **тип** x. Результат – целочисленное значение типа. Например, если $x = 1$, то **тип** x = **цел64**.

Имеются удобные операции проверки объекта на принадлежность заданному классу типов или на заданный формат: **естьцел**, **естьвещ**, **естьнабор**,

естьпусто, **естьф32**, **естьф64**, **естьф128**. Например, выражение **естьвещ** x истинно, если x – вещественное одного из трех возможных форматов; **естьпусто** x истинно, если x равен **пусто32** или **пусто64**; **естьф128** x истинно – если x – вещественное 128. Проверку на тип "набор" рекомендуется выполнять операцией **естьнабор**, а не выделением типа (см. п. 1.4).

Операция ⇔ (логическое равенство) – обобщение обычного равенства – определена для всех типов и проверяет совпадение и тегов, и значений, т.е. совпадение объектов. Например, x ⇔ x всегда истинно (если x не имеет побочных эффектов), в частности, **пусто64** ⇔ **пусто64**, так как информационная часть пустого объекта равна нулю, а **пусто32** ⇔ **пусто64** всегда ложно.

1.4. Наборы

Описание простых типов данных языка Эль-76 начнем не с чисел (более традиционных), а с наборов, так как наборы – наиболее простые типы, структура которых определяет "систему координат" для представления всех остальных типов.

Набор – это последовательность смежных битов (элементов), размещаемая в одном слове и имеющая свой тег. Длина набора (число элементов) не превосходит 64 – длины слова. Набор длины 1 (одноэлементный) соответствует логическому значению, длины 8 – символу, длины 4 – шестнадцатиричной цифре. Набор длины 64 называется *полным*, в остальных случаях – *неполным*. Отдельных типов данных "логическое" и "символ" в Эль-76 нет, благодаря наборам в них нет необходимости. Другие функции наборов рассмотрены ниже.

1.4.1. Внутреннее представление наборов изображено на рис. 1. Набор занимает целое слово независимо от длины. Элементы набора занимают младшие разряды слова (наиболее удобно представлять их себе "прижатыми" к правому краю слова) и нумеруются справа налево от 0 до 63.

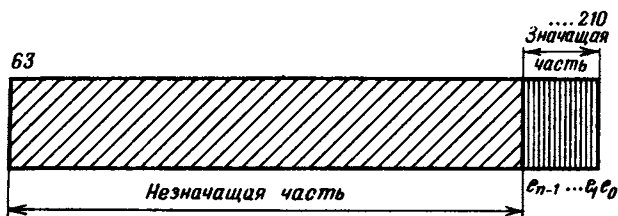


Рис. 1

Элементы набора образуют *значащую часть*. Остальная (*незначащая*) часть слова содержит информацию о длине, но при выполнении операций над набором обнуляется.

1.4.2. Изображение наборов. Значащая часть набора задается последовательностью двоичных, восьмеричных или шестандцатиричных цифр либо символов, взятой в кавычки (используется символ "двойная кавычка"). Изображению может предшествовать слово **наб**. Например, один и тот же

набор длины 24 может быть задан следующими изображениями:

1"110000011100001011000011"
3"60341303"
4"C1C2C3"
8"ABC"
"ABC"
наб "ABC"

Цифра в начале изображения указывает способ представления: 1 – двоичными цифрами, 3 – восьмеричными цифрами, 4 – шестандцатеричными цифрами, 8 – символами (цифру 8 в начале изображения можно опускать). В системе "Эльбрус" для кодировки символов применяется стандартный код ДКОИ-8. Разрешается комбинировать различные способы представления в одном изображении, например 4"C1"8"BC" – тот же набор.

Для логических значений введены особые изображения: **истина** (эквивалентно 1"1") и **ложь** (эквивалентно 1"0").

1.4.3. Битовое представление. Поля. Информационную часть любого простого значения можно рассматривать как полный набор (вещественное 128 – как два полных набора). Именно такое *битовое представление* (в сочетании с тегом) используется при выполнении операций. Элементы в нем нумеруются так же, как в наборах. Значение формата **ф32** преобразуется в битовое представление следующим образом: целое 32 (без знака) размещается в младших разрядах слова, его знак – в старшем разряде, вещественное 32 – в старшей половине слова; пусто 32 – в младшей, остальные разряды равны нулю.

Поле – это последовательность смежных битов слова. В языке Эль-76 поле имеет стандартное обозначение [Н : К], где Н – *начало* (номер старшего бита), К – *количество* битов. Н и К могут быть любыми выражениями. Например:

[63 : 1] – знак числа в битовом представлении (можно писать [63]),
[7 : 8] – младший байт слова (можно писать [7 :]).

Для удобства введена возможность указания значений Н и К в более крупных единицах – цифрах (**ф4**) и байтах (**ф8**). Например, обозначения [31 : 32], [**ф4** 7 : 8] и [**ф8** 3 : 4] задают одно и то же поле.

Наиболее удобно использовать в программе не явное обозначение поля, а его идентификатор, введенный с помощью *описания поля*:

```
поле
    знак = [63],      % знак числа,
    байт = [7:];      % младший байт слова
```

Язык Эль-76 позволяет свободно работать с полями: выделить поле из любого значения, изменить поле в значении, не принадлежащем классу адресной информации, полностью сформировать любое такое значение по полям.

Выделение поля выполняется операцией "поле значения", форма записи которой аналогична операции выборки поля структуры во многих языках программирования: *х.поле*, где *х* – значение, заданное первичным выражением (например, идентификатором, см. п. 1.10), *поле* – обозначение поля (явное обозначение или идентификатор). Результат операции – *набор*, об-

р[illegible]ый значениями битов указанного поля. Например, если x = "ABC", те[illegible]3 : 8] ⇔ "A", x. [15 : 8] ⇔ "B", x. [7:] ⇔ "C". Если x = −1, то x.[illegible] ⇔ 1"1", а x. [31 : 1] ⇔ 4"00000001".

[illegible]*ирование* – операция, обратная выделению поля. Выполняет сборк[illegible]ения из его типа и полей. Например, результатом операции формир[illegible]

[illegible] : **цел64**, [63] : 1, [62:] : 2)

[illegible]ется число −2 типа **цел64**. В операции может быть задана основа фор[illegible]ования, в которую производятся заданные вставки. Например, значе[illegible] выражения

x. ([63] : 0)

– модуль числа x. По умолчанию основа равна нулю, указание типа не обязательно (по умолчанию – **цел64**). Попытка формирования значений из класса адресной информации приводит к ситуации "привилегированные действия".

Изменение поля выполняется с помощью конструкции "поле переменной" и присваивания. Операция

x.знак := 0

обнуляет знак числа x.

1.4.4. Операции над наборами. Набор – самый многофункциональный тип данных в языке Эль-76. В зависимости от его длины и от операции он используется как:

– число типа **цел64** без знака (в арифметических операциях),

– логическое значение (одноэлементный набор – в логических операциях),

– символ (набор длины 8 – в операциях над строками),

– строка битов, цифр или байтов (в операциях над строками),

– двоично-десятичное число (в операциях перевода чисел).

В *арифметических операциях* набор эквивалентен целому 64 без знака (при этом бит [63] набора должен быть равен нулю, иначе – прерывание). Например, значение выражения

4"A" + 4"B"

равно 21 (числу типа **цел64**).

Логические операции – **и** (конъюнкция), **или** (дизъюнкция), **не** (отрицание), **экв** (эквивалентность) – выполняются поразрядно над наборами любой длины. Операнды и результат приводятся к наибольшей длине. Применение этих операций к одноэлементным наборам дает логические операции в традиционном смысле слова. Приоритеты операций (в порядке убывания): **не**, **и**, **или**, **экв**. Например, формула

4"AA" **и** 4"BB" **или не** 4"C" **экв** 4"DDD"

имеет значение 4"289". Поскольку формулы в языке Эль-76 исполняются слева направо с учетом приоритетов, то порядок выполнения операций в этом примере следующий: **и**, **не**, **или**, **экв**.

Подчеркнем еще раз, что логические операции применимы только к наборам, попытка применения их, например, к числам (1 **или** 2) приводит к прерыванию.

Операции отношения = (равно) < > (не равно), < (меньше), < = (меньше или равно), >= (больше или равно), > (больше) выполняются над наборами как над неотрицательными целыми числами, но при этом учитываются все 64 бита. Например, выражение

4”8000000000000001”> 1”1”

истинно (для чисел с такими же битовыми представлениями имело бы место отношение <, так как первый операнд является битовым представлением числа –1, а второй – числа 1).

Операции над битовым представлением **пче** (подсчет числа единиц) и **перв1** (номер первой единицы) вычисляют, соответственно, количество единиц в битовом представлении набора и номер старшего бита, равного единице. Эти операции удобны для реализации операций над множествами: если множество представлено короткой битовой шкалой (набором), то операция **пче** вычисляет мощность множества, **перв1** – максимальный номер элемента множества. Например:

пче 1”1001” = 2, **перв1** 1”1001” = 3, **пче** 1”000” = 0
перв1 1”000” = –1.

Другие операции, в которых могут участвовать наборы, рассматриваются в п. 1.5 и 1.6.

1.5. Числа

Арифметика языка Эль-76 имеет как традиционные черты, так и ряд особенностей, связанных с ориентацией системы на языки высокого уровня. Основная особенность – полиморфизм арифметических операций, которые применимы к объектам любого числового типа и выдают результат наиболее общего типа и наибольшего формата.

В языке имеются целые числа форматов **ф32** и **ф64** и вещественные числа форматов **ф32**, **ф64** и **ф128**. В памяти и при выполнении операций числа представлены в прямом коде (внутреннее представление отрицательного числа отличается одним битом в разряде знака).

1.5.1. Изображение чисел. Основной способ – традиционное десятичное изображение (порядок выделяется символом *е*), дополненное следующими возможностями:

– явное указание типа и формата числа (по умолчанию число имеет формат **ф64**),

– изображение вещественного числа с помощью целого, с явным указанием вещественного типа.

Примеры:

0 1 – целые 64,
3.14 – вещественное 64,
цел32 125 – целое 32,
вещ32 2.71828 – вещественное 32,
вещ128 1'– вещественное 128.

Числа можно также изображать совокупностью числового типа и *представлением набора* (п. 1.4.2), задающего битовое представление числа. В качестве примера приведены изображения максимальных числовых

значений всех типов и форматов:

цел32 4"7FFFFFFF" (**цел32** 2147483647) – целое 32

цел64 4"7FFFFFFFFFFFFFFF"

(9223372036854775807) – целое 64

вещ32 4"3FFFFFFF00000000"

(**вещ32** 7.2370056*e*75) – вещественное 32

вещ64 4"3FFFFFFFFFFFFFFF"

(7.2370055773322621*e*75) – вещественное 64

вещ128 4"3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"

(**вещ128** 1.2522062065042790406119202197251 57*e*19727) – вещественное 128.

Таким образом, максимальный допустимый порядок вещественных чисел формата **ф32** и **ф64** равен 75, формата **ф128** – 19727.

Необходимые пояснения:

– вещественное 32 в битовом представлении хранится в виде [63 : 32],

– в изображении вещественного 128 представление набора задает все 128 разрядов.

1.5.2. Арифметические операции "+", "–", "*", "/" применимы к числам любого типа и формата. Перед операцией операнды преобразуются к наиболее общему типу и к наибольшему формату, и результат получается этого типа и формата. Если результат не представим в виде целого числа заданного формата, то он преобразуется в вещественное число. Результат операции деления "/" всегда вещественный. Например:

вещ32 1.0 + 2 равно 3.0,

цел32 1000000000 * **цел32** 1000000000 равно **вещ32** 1*e*18,

1 / 2 равно 0.5..

Операция **умнд** (умножение с двойной точностью), в отличие от операции *, "удваивает" формат результата, если формат операндов меньше **ф128**. Например:

2.0. **умнд** 2 равно **вещ128** 4.0,

цел32 3 **умнд цел32** 5 равно **цел64** 15.

Операция возведения в степень **. Показатель степени должен быть константным выражением (1.9.2) целого типа. Например, если *c* – целая константа, $v1$ и $v2$ – переменные, то допустимы формулы $v1$ ** 3, $v1$ ** *c*, $v1$ ** –*c*, а формула $v1$ ** $v2$ ошибочна.

1.5.3. Арифметические операции над целыми числами. Операции "+:", "–:", "*:", "/:" и **остат** составляют группу операций *целочисленной арифметики.* Их операндами могут быть только целые числа или наборы, результатом – целые числа (при переполнении целых – прерывание). Операция "/:" выполняет деление нацело, **остат** – взятие остатка от деления. Результаты этих операций для любых целых чисел *а* и *в* связаны соотношением

а = (*а* /: *в*) * *в* + (*а* **остат** *в*).

Результат операции *а* **остат** *в* имеет знак *а*. Например, 7 /: 2 = 3, 7 **остат** 2 = 1, –7 **остат** 2 = –1. Выполнение операции *а* **остат** *в*, где *а* = 1 и *в* = 0, приводит к прерыванию.

1.5.4. Операции отношения =, < >, <, < =, > =, > определены для любых чисел. Как и в арифметических операциях, операнды приводятся к наиболее общему типу и наибольшему формату. Результат – одноэлементный набор. Например, для любых целых чисел x и y истинно значение выражения

$x < y$ **или** $x >= y$

В этом выражении сначала выполняются операции отношения < и > =, затем – логическая операция **или**.

1.5.5. Особенности присваивания чисел. При присваивании числа (набора) переменной меньшего формата выполняется аппаратное приведение значения к формату переменной (округление). Если присваиваемое значение – целое, то выполняется преобразование в целое, а если это невозможно – в вещественное. При невозможности приведения к меньшему формату – прерывание.

П р и м е р.

Пусть имеются описания переменных

ф32 x; **ф64** y

Тогда операция

x := 3.14

выполняет присваивание переменной x значения **вещ32** 3.14. При присваивании

y := **вещ128** 1.0

переменная y получит значение **вещ64** 1.0. Присваивание

x := 1 (или x := 1"1")

устанавливает значение x равным **цел32** 1, а присваивание

x := 1000000000000

– равным **вещ32** 1*e*12. Присваивание

x := **вещ128** 1*e*100,

приводит к прерыванию.

1.5.6. Операции преобразования чисел. Совокупность операций **целокр, целобр, вещокр, вещобр, ф32окр, ф32обр, ф64окр, ф64обр, цел64окр, цел64обр** и **в128** служат для преобразований чисел любого типа и формата. Назначение этих операций отражено в мнемонике их назначений: *цел* – преобразование в целое, *окр* – с округлением, *обр* – с обрубанием (отбрасыванием дробной части), *ф32* (*ф64, ф128*) – преобразование к указанному формату с сохранением типа. Например, **цел64обр** x – преобразование числа x в целое 64 с обрубанием (аналог функции trunc языка Паскаль). **ф32окр** x – в число формата **ф32** с округлением (целое – в целое, вещественное – в вещественное). **в128** x – преобразование в вещественное 128.

1.5.7. Операции перевода чисел. В МВК "Эльбрус" имеются операции перевода чисел из символьной во внутреннюю форму и обратно. Перевод выполняется в два этапа:

строка символов → цифровой набор → целое.

Здесь под цифровым набором понимается набор длины, кратной 4, рассматриваемый как последовательность двоично-десятичных цифр. Для

указания знака при переводе служит системная логическая переменная — *признак* (*триггер*) **тго**. Операции **бит** и **дес** выполняют преобразование цифрового набора в целое и обратно, преобразование *строка* ⟷ *цифровой набор* выполняется операциями над строками **пак** и **распак**, см. п. 1.6.6.

Операция **бит** x преобразует цифровой набор x в целое 64. Знак числа берется из признака отношения. Обратная операция **дес** x преобразует целое число x в цифровой набор. Знак запоминается в признаке отношения. Если число x имеет более 16 значащих цифр, то прерывание.

Признак отношения может быть установлен явно операцией **тгз** y, которая записывает в **тго** знак целого числа y.

Примеры:

дес цел64 1"1111011" = 4"123" (**тго** устанавливается в 0);

дес цел64 4"8000000000000001" = 4"1" (**тго** устанавливается в 1).

Если выполнена операция **тгз** 1 (**тго** установлен в 0), то

бит 4"00123" = **цел64** 1"1111011",

бит 4"0" = 0.

Если выполнена операция **тгз** −1 (**тго** установлен в 1), то

бит 4"0012345" = −12345.

Внутреннее представление чисел описано в работе [52].

1.6. Векторы

Вектором в системе "Эльбрус" называется одномерный массив элементов, расположенных смежно в области математической памяти (полное название — *смежный вектор*, или *с-вектор* [52]). Вектор — простейший структурированный объект, операции над которым в языке Эль-76 имеют аппаратную поддержку. Ввиду динамизма системы "Эльбрус" в одном векторе можно хранить значения как одного, так и разных типов и форматов.

Векторы могут быть *локальными* (существующими только при исполнении некоторого блока или процедуры) и *глобальными* — существующими до окончания программы (п. 1.18).

Память для любого вектора выделяется при первом обращении к нему. Вектор большой длины разбивается на страницы и может храниться в оперативной памяти не полностью, но этот факт остается незаметным для программиста.

Векторы могут иметь постоянную или переменную длину не более $2^{20} - 1 = 1048575$ слов. Для представления динамической структуры данных (например, списка), во избежание фрагментации памяти, рекомендуется использовать вектор переменной длины, в котором размещаются элементы списка, а не представлять каждый элемент отдельным вектором.

1.6.1. Указатель, формат и генератор вектора. В операциях вектор задается *указателем* — простым объектом типа **деск**. Указатель содержит *формат* элементов вектора, его *длину* (число элементов в единицах формата), *признаки защиты* и *начальный адрес*. Элементы вектора нумеру-

ются начиная от нуля. Удобно представлять их расположенными слева направо (в отличие от набора, в котором элементы справа налево).

Векторы могут быть следующих форматов: **ф1** (*бит*), **ф4** (*цифра*), **ф8** (*байт*), **ф32** (*полуслово*), **ф64** (*слово*) и **ф128** (*два слова*). Векторы форматов **ф32**, **ф64** и **ф128** называются *простыми векторами*, форматов **ф1**, **ф4** и **ф8** – *строками*.

Способ создания нового вектора в языке Эль-76 отличается от традиционного описания массива:

var *a* : **array** [1 . . *n*] **of real**

Такое описание постоянно связывает идентификатор *a* с массивом в некоторой процедуре или блоке. В языке Эль-76 для создания вектора введена более общая конструкция – *генератор вектора*:

лок вект [*n*] **ф64**

Результат этой конструкции – указатель локального вектора формата **ф64** и длины *n*. Длина вектора *n* может быть любым выражением. Вместо формата **ф64** может быть задан любой другой формат – **ф1**, **ф4**, **ф8**, **ф32**, **ф128**. Служебное слово **глоб** (вместо **лок**) задает генерацию глобального вектора. Максимальная длина вектора зависит от формата и равна соответственно: для форматов **ф32**, **ф64** и **ф128** – 1048575 (2 ** 20–1), **ф8** – 524287 (2 ** 19–1), **ф4** – 262143 (2 ** 18–1), **ф1** – 65535 (2 ** 16–1).

Элементам вектора формата **ф64** присваиваются начальные значения **пусто64**, формата **ф32** – значения **пусто32**. Векторы остальных форматов заполняются значениями **пусто64**.

Конструкция, эквивалентная традиционному описанию массива, получится, если поместить генератор массива в правой части описания константы:

конст *a* = **лок вект** [*n*] **ф64**

Возможен и другой вариант – с помощью описания переменной с начальным присваиванием:

ф64 *a* := **лок вект** [*n*] **ф64**

Первый вариант предпочтительнее, так как он обеспечивает дополнительный контроль при трансляции: присваивания идентификатору *a* запрещены. Ошибочное присваивание (например, *a* := *в*) может появиться в программе как неверное написание операции *пересылки* (поэлементного присваивания) векторов (п. 1.6.4). Разумеется, постоянное значение имеет только идентификатор *a* (указатель вектора), а значения *элементов* вектора *a* можно изменять.

Второй вариант описания рекомендуется, если переменную *a* предполагается использовать для ссылки либо на вектор переменной длины, либо на фрагменты (подмассивы) одного вектора. В описании для переменной *a* задан формат **ф64**, так как указатель вектора (как отдельный объект) имеет этот формат. Второе вхождение слова **ф64** в данном случае относится к генератору и задает формат элементов вектора.

1.6.2. Векторы переменной длины. Генератор вектора вида

лок вект [*m* : *n*] **ф64**

создает *вектор переменной длины.* m – *начальная длина* вектора, n – *максимальная* (m и n – выражения). Длину такого вектора можно увеличивать до n элементов стандартной функцией *ОС*,

измдлину (*в*, d),

где *в* – указатель вектора переменной длины, d – приращение. Функция *измдлину* выдает указатель вектора *в* с измененной длиной. Отметим, что операция изменения длины имеет поддержку аппаратуры и ОС, обеспечивающую одинаковую эффективность обращений к векторам переменной и постоянной длины. В частности, изменение длины вектора не связано с его переписью на новое место в памяти. Например:

ф64 *в* := **лок вект** [100 : 1000] **ф64**;

% начальная длина – 10 слов, максимальная – 1000 слов

в := *измдлину* (*в*, 200) % текущая длина – 300 слов

Другая форма вызова функции *измдлину*:

измдлину (**имя** *в*, 200)

Здесь присваивание переменной *в* указателя вектора с увеличенной длиной выполняется при исполнении функции *измдлину*. Этот указатель является результатом функции.

1.6.3. Основные операции над векторами. Операция **длина** вычисляет длину вектора (в виде целого 64). Например, для вектора a из п.1.6.1 **длина** $a = n$. Эту операцию рекомендуется использовать для указания пределов изменения индекса в цикле (от 0 **до** **длина** $a - 1$) при последовательной обработке вектора a.

Операция "элемент массива" служит для обращения к элементу массива или изменения его значения. Наиболее часто используется ее форма, принятая в большинстве языков программирования: *в* [i], где *в* – указатель вектора (первичное), i – индекс (последовательное предложение со значением типа "целое" или "набор"). Значение i должно удовлетворять условию

0 < = i и i < **длина** *в*

иначе возникает ситуация "граница массива".

Пример.

x [i] := **если** i < **длина** y **то** y **иначе** z **все** [i]

Здесь i-му элементу вектора x присваивается либо значение i-го элемента вектора y, либо значение i-го элемента вектора z.

Операция "*подмассив*" аналогична конструкции "вырезка" языка Алгол-68, но имеет несколько больше возможностей. Она предназначена для получения указателя на любой отрезок вектора либо указателя на тот же вектор (или его отрезок) как на вектор другого формата. Таким образом, формат – характеристика конкретного указателя, а не самой области памяти.

Наиболее употребительная форма операции "*подмассив*":

в [*н* : *к*]

Здесь *в* – указатель вектора, *н* – начальный индекс, *к* – количество (в отличие от Алгола-68, где *к* – конечный индекс). Результат операции –

указатель на подмассив вектора *в*, начинающийся с *н*-го элемента, длиной *к* элементов. Элементы в подмассиве нумеруются также начиная от нуля (до *к* –1). Форма *в* [:*к*] эквивалентна *в* [0 : *к*] (по умолчанию – с начала), форма *в* [*н*:] – форме *в* [*н* : **длина** *в*–*н*] (по умолчанию – до конца).

Подчеркнем, что операция "подмассив" не приводит к созданию новой области памяти, а лишь формирует новую ссылку на часть исходной области. Например, если выполнена последовательность операторов:

с := *в* [1 : 5];
с [0] := 0

то значение *в* [1] также становится равным нулю.

Другой вариант операции "подмассив" – с изменением формата: *в* [**ф32** *н* : *к*] (вместо **ф32** может быть задан любой формат). Чаще используется форма *в* [**ф32**:], результатом которой является указатель того же вектора *в*, но другого формата.

Пример.

Пусть *в* – указатель вектора формата **ф64** длины 10. Тогда:

1. *с* := *в* [4 : 5] – указатель вектора из 5 элементов: *с* [0] ~ *в* [4], *с* [1] ~ *в* [5], *с* [2] ~ *в* [6], *с* [3] ~ *в* [7], *с* [4] ~ *в* [8];

2. *в* [:3] – указатель вектора из трех элементов: *в* [0], *в* [1], *в* [2];

3. *в* [**ф8**:] – указатель вектора *в*, рассматриваемого как байтовая строка длиной 80;

4. *d* := *в* [**ф32** 1 : 3] – указатель подмассива *в* формата **ф32** и длины 3: *d* [0] ~ *в* [**ф32** 1], *d* [1] ~ *в* [**ф32** 2], *d* [2] ~ *в* [**ф32** 3].

1.6.4. Групповые операции над простыми векторами. *Групповые операции* – это операции последовательной поэлементной обработки векторов. В МВК "Эльбрус-1" и МВК "Эльбрус-2" они реализованы аппаратно. Имеются два класса групповых операций: над *простыми векторами* формата **ф64** и над *строками*. Групповые операции над строками рассмотрены в п. 1.6.6.

Пересылка простых векторов – операция их *поэлементного присваивания*. Присваивание массивов имеется во многих языках программирования, но в языке Эль-76 оно имеет более широкие возможности – пересылка в один вектор содержимого нескольких векторов, заполнение вектора одинаковыми значениями. В наиболее простом случае операция пересылки простых векторов имеет вид

а < := **ф64** *в*

где *а* и *в* – указатели векторов формата **ф64**. Такая операция называется простой пересылкой. Она выполняет поэлементное присваивание *а* [0] := := *в* [0], *а* [1] := *в* [1] и т.д. При исчерпании одного из векторов операция заканчивается. Причина окончания указывается в стандартных логических переменных – *признаках*: **тти** – *признак источника* (*в*), **ттп** – *признак получателя* (*а*). Перед началом операции оба признака устанавливаются в **ложь**, а после ее окончания соответствующий признак (или оба сразу) принимает значение **истина**. Результат операции – указатель на необработанную часть вектора *а*. Элементы вектора *в* пересылаются вместе с тегами.

П р и м е р.

Пусть выполнены следующие описания и операторы:

ф64 *а* := **лок вект** [5] **ф64**

в := **лок вект** [3] **ф64**;

в [0] := 1, *в* [1] := 3.14; *в* [2] := "ABC"

Тогда:

1. После выполнения операции *а* < :=**ф64** *в*: *а* [0] ⇔ 1, *а* [1] ⇔ 3.14, *а* [2] ⇔ "ABC"; остальные элементы *а* – пустые объекты, тгп ложно, тги истинно.

2. После выполнения операции *а* [:2] < :=**ф64** *в*: *а* [0] ⇔ 1, *а* [1] ⇔ 3.14, остальные элементы *а* – пустые объекты, тгп истинно, тги ложно.

3. После выполнения операции *а* [:2] < :=**ф64** *в* [:2] эффект тот же, но оба признака истинны.

Форма операции простой пересылки с несколькими источниками

а < :=**ф64** *в1* && *в2* && ... && *вк*

эквивалентна последовательному выполнению операций

((... (*а* < :=**ф64** *в1*) < :=**ф64** *в2*) < :=**ф64** ...) < :=**ф64** *вк*

т.е. выполняет пересылку в вектор *а* последовательно элементов векторов *в1*, . . ., *вк*.

Операция заполнения

а < :=**ф64** **заполн** *x*

присваивает всем элементам вектора *а* одно и то же значение *x*.

В операции пересылки можно указать максимальное количество элементов *к*:

а < :=**ф64** *в* **длиной** *к*.

В этом случае операция пересылает не более *к* элементов и завершается по исчерпанию либо *а*, либо *в*, либо *к*. В последнем случае признаки тгп и тги принимают значение ложь.

П е р е с ы л к а с м о д и ф и к а ц и е й. При последовательной обработке векторов часто требуется выполнять какие-либо действия с необработанной частью источника или получателя либо определить количество обработанных элементов. При этом удобно использовать вариант операции пересылки с модификацией:

мод *а* < :=**ф64** **мод** *в* **длиной мод** *с*.

По окончании этой операции значения переменных *а* и *в* модифицируются "продвинутыми" указателями, значение переменной *с* уменьшается на число обработанных элементов.

П р и м е р (продолжение предыдущего примера).

Пусть, в дополнение к описаниям векторов *а* и *в* и инициализации *в*, выполнены следующие описания и операторы:

ф64 *с* := **лок вект** [2] **ф64**,

d := **лок вект** [10] **ф32**,

с [0] := 2; с [1] := 3

Тогда:

4. После выполнения операции

а< :=**ф64** *в* [:2]&&*с* && **заполн** 0

а [0] ⇔ 1, *а* [1] ⇔ 3.14, *а* [2] ⇔ 2, *а* [3] ⇔ 3, *а* [4] ⇔ 0, **тгп** истинно, **тги** ложно.

5. После выполнения операции

а< :=**ф64 заполн** 0 **длиной** 4&& **мод** *с*

а [0], *а* [1], *а* [2] и *а* [3] равны нулю, *а* [4] ⇔ 2, **тгп** истинно, **тги** ложно, переменной *с* присвоено значение *с* [1 :].

6. Операция

а< := **ф64**

приводит к прерыванию (формат *d* – не **ф64**). Правильное написание:

а< := **ф64** *d* [**ф64** :]

Поиск по простому вектору. Групповая операция поиска охватывает распространенные случаи поиска по вектору, например линейный поиск (вперед и назад).

Операция позволяет найти элемент, равный заданному эталону либо находящийся с эталоном в некотором отношении (<, >, < =, > =, <>).

Пример.

Поиск в векторе *в* первого от начала элемента, равного *х* (тип элемента игнорируется):

```
к:= поиск (в, 0) до = х;
если тги
то запф (печ, :"элемента нет!"/)
иначе запф (печ, стр8"индекс = ", к)
все
```

Операнды операции **поиск** – указатель вектора *в* и начальной индекс 0. Результат – индекс найденного элемента. Если элемент не найден, то признак **тги** принимает значение **истина**, а результатом является индекс, вышедший за границу вектора. Если вместо слова **до** задано слово **вниздо**, а начальный индекс равен (**длина** *в* – 1), то будет найден первый от конца элемент, равный *х*.

1.6.5. Изображение векторов. Константные простые векторы и строки с постоянными элементами задаются в программе на Эль-76 *изображениями массивов констант* и *изображениями строк*. Массивы констант и строки размещаются в особой области математической памяти, общей для всех задач (т.е. для всех использующих их задач – в одном экземпляре). Указатели их содержат признаки защиты от записи. Попытка изменения значений их элементов приводит к ситуации "нарушение защиты".

В изображении массива констант задается требуемый формат указателя вектора и значения его элементов (константные выражения). Для обозначения повторяющихся значений элементов используются *повторители*. Например: **мконст ф64** ("пн", "вт", "ср", "чт", "пт", "сб", "вс").

Результат конструкции "изображение массива констант" – указатель вектора заданного формата (**ф64**). Его можно обозначить идентификатором с помощью описания константы:

конст экран = **мконст ф8** (/3 * 79/"   ")

% строка для чистки экрана дисплея в диалоговой системе,

% см. п. 1.26

Изображение строки записывается аналогично изображению набора (п. 1.4.2) – в виде последовательности двоичных, восьмеричных, шестнадцатеричных цифр или символов. Формат строки задается словом **стр1**, **стр4** или **стр8**. Например:

конст *с* = **стр8** "длинная строка",
буквицифр = **стр1** 4"00000000000000000000000000000000"
4"00000000000000BB7FDA7FCC3FDBFFFE"

Первое изображение обозначает байтовую строку длиной 14, второе – битовую строку длиной 256, описывающую множество всех кодов букв и цифр в кодировке ДКОИ-8. В примере показан также способ переноса длинного изображения строки на новую строку текста.

Короткие последовательности битов, цифр или символов (длиной не более слова) рекомендуется изображать в виде *наборов*, так как это не связано с выделением памяти в массиве констант. Например, рекомендуется писать "*авс*" вместо **стр8** "*авс*". В операциях над строками набор и строка эквивалентны.

1.6.6. Групповые операции над строками. *Строки* – особый класс векторов, предназначенный для хранения информации, упакованной в виде последовательности битов, шестнадцатеричных цифр или байтов. Однако к ним, как и к простым векторам, применимы все основные операции (длина, элемент массива, подмассив). Рассмотрим групповые операции над строками, учитывающие их специфику. Большинство из них применимо только к байтовым строкам и предназначено для символьной обработки. Исключение составляют две общие операции над строками – *простая пересылка* и *сравнение*.

Простая пересылка строк. Семантика этой операции аналогична пересылке простых векторов (п. 1.6.4). Форма записи

а < := *в*,

где *а* и *в* – строки одинакового формата (**ф1**, **ф4** или **ф8**). Пересылка выполняется слева направо (от начала к концу), поэлементно, до исчерпания получателя *а* (при этом признак **тп** принимает значение **истина**) или источника *в* (значение **истина** принимает признак **ти**). Как и для пересылки простых векторов, возможно указание нескольких источников, заполнение, модификация, ограничение количества элементов. Например, если

ф64 *а* := **лок вект** [10] **ф8**,
в := **лок вект** [5] **ф8**

то после выполнения операции

мод *а* < := в && **заполн** " " **длиной** 3

а [0] = *в* [0], *а* [1] = *в* [1], ... , *а* [4] = *в* [4], *а* [5] = *а* [6] = *а* [7] = " ", **тп** и **ти** ложны, переменная *а* принимает значение *а* [7 :].

Сравнение строк. Над строками любого формата определены операции сравнения =/, < > /, < /, < =/, > = /, > / (знак операции образован из знака операции отношения добавлением символа «/»). Для строк разных форматов эти операции выполняются по-разному, в соответствии с общепринятым смыслом операции сравнения для представляемых ими объектов: для байтовых и цифровых строк – *лексикографическое сравнение,* для битовых строк – *равенство и включение множеств* (например, операции < =/ и > =/ соответствуют нестрогим включениям множеств $\subseteq$ и $\supseteq$). Следующие выражения имеют значение **истина**:

стр8 "алфавит" </ **стр8** "алхимик"
стр4 "авс2" > / **стр4** "авс1"
стр1 "10010" < =/ **стр1** "110111"

При сравнении байтовых строк следует учитывать, что сравнение символов выполняется в соответствии с их упорядочением в кодировке ДКОИ-8.

Операции сравнения, как и другие групповые операции, выполняются поэлементно, до исчерпания одной из строк. Правый операнд может быть набором и рассматривается как строка. Для строк формата **ф4** и **ф8** результат сравнения определяет первая пара неравных элементов, для строк формата **ф1** отношение считается выполненным, если оно истинно для *всех* пар элементов. Если же все элементы оказались равны, то считается, что отношения =/, < =/, > =/ выполняются, остальные – нет. В связи с этим операция лексикографического сравнения "*а* меньше *в*" для строк *а* и *в* произвольной длины записывается более сложно:

если длина *а* > = **длина** *в*
то *а* < / *в*
иначе *а* < =/ *в*
все

Результат сравнения сохраняется в признаке отношения **тго**.

Операции символьной обработки. К ним относятся: различные формы *условной пересылки, поиск по строке* (сканирование), *проверка принадлежности множеству, упаковка* и *распаковка.* Эти операции применимы только к байтовым строкам. Совокупность операций символьной обработки языка Эль-76 соответствует базовому набору действий, выполняемых при лексическом анализе текста: пропуск ограничителей, выделение лексем (слов), перевод чисел во внутреннюю форму и т.п. Операции рассматриваются на примерах решения типичных задач лексического анализа. В примерах предполагается, что *текст* – текущий указатель по обрабатываемому входному тексту (байтовой строке). По мере просмотра текста указатель *текст* модифицируется. Кроме того, имеется буфер *буф* для набора очередной лексемы.

Пересылка с проверкой отношения. Пусть текст состоит из слов, разделенных одним или несколькими пробелами. Требуется выделить в буфер *буф* очередное слово текста. На языке Эль-76 эта задача решается одной операцией *пересылки с проверкой отношения:*

буф < := **мод** *текст* **пока** < > " "

Операция заканчивается либо при исчерпании одного из операндов (при этом устанавливается признак тгп или тги), либо при обработке символа источника, не удовлетворяющего отношению (т.е. пробела). В последнем случае этот символ не пересылается, и устанавливается признак отношения тго.

В качестве отношения может быть задан любой из шести символов операций отношения, в качестве эталона для сравнения – любой набор. Если, например, пробелы заданы в упакованном виде (что типично для текстовых файлов МВК "Эльбрус", п. 1.27), т.е. k подряд идущих пробелов ($k < 64$) заменены одним байтом со значением k, то в операции следует использовать другой знак отношения:

буф < := **мод** *текст* **пока** > " "

Если требуется также вычислить длину слова (*дл*), то используется форма операции с модификацией длины (*длбуф* – длина буфера):

дл := *длбуф* ;
буф < := **мод** *текст* **длиной мод** *дл* **пока** < > " ";
дл := *длбуф* – *дл*

Пересылка по шкале. Усложним предыдущую задачу. Пусть слова в тексте могут разделяться не только пробелами, но и знаками препинания ".", ",", ":", "; ","–". Тогда для выделения слова (последовательности символов от ограничителя до ограничителя) следует воспользоваться другой формой условной пересылки – *пересылкой по шкале*. Ее использование требует указания множества символов. Для этого применяется традиционный способ: множество символов представляется *битовой шкалой* – строкой формата **ф1** и длиной 256 (по числу возможных кодов символов). Единица в i-м элементе шкалы означает, что символ с кодом i принадлежит заданному множеству. Шкала обычно задается в программе в виде описания константы, но сформировать ее можно автоматически и затем распечатать или вывести в текстовой файл, например:

конст *огран* = **лок вект** [256] **ф1**;
огран [**ф64**:] < : =**ф64 заполн** 0;
огран [" "] := *огран* [" . "] := *огран* [", "] := 1;
огран [" : "] := *огран* ["; "] := *огран* ["–"] := 1

Операция пересылки по шкале, решающая описанную задачу, записывается в следующей форме:

буф < := **мод** *текст* **пока не среди** *огран*

Вариант с модификацией длины:

буф < := **мод** *текст* **длиной мод** *дл* **пока не среди** *огран*

Порядок операндов фиксирован: получатель, источник, длина, эталон.

Другой пример использования пересылки по шкале – выделение идентификатора из текста (задача, типичная для разработки трансляторов). Пусть *буквцифр* – шкала, описывающая множество всех букв и цифр (см. пример в п. 1.6.5). Идентификатор выделяется из текста и пересылается в буфер с помощью следующего варианта операции пересылки

(для простоты вычисление длины опущено):

буф < := **мод** *текст* **пока среди** *буквицифр*

Сканирование (поиск по строке). Вернемся к задаче анализа текста. Пусть очередное слово (до ближайшего ограничителя) выделено в буфер. Требуется пропустить ограничители перед следующим словом. Эта задача также решается с помощью одной операции – поиска по строке. Вариант с ограничителями-пробелами:

мод *текст* **от** < > " "

Вариант с ограничителями – упакованными пробелами:

мод *текст* **от** > " "

Вариант с ограничителями-пробелами и знаками препинания:

мод *текст* **от не среди** *огран*

Наконец, приведем решение задачи пропуска идентификатора во входном тексте (с вычислением его длины):

дл := *дл1* := **длина** *текст;*
мод *текст* **от длиной мод** *дл* **либо не среди** *буквицифр;*
дл := *дл1* – *дл*% *дл* – длина идентификатора

Символ **от** в операции сканирования означает, что выделяется часть строки начиная от первого символа, удовлетворяющего заданному отношению.

Во всех случаях операция заканчивается либо при исчерпании текста-источника (устанавливается признак **тги**), либо при обнаружении символа, удовлетворяющего заданному отношению (устанавливается признак **тго**). Указатель *текст* настраивается на текст, начинающийся с найденного символа.

Пересылка с переводом выполняет перекодировку текста из одного 8-разрядного кода в другой в процессе его пересылки. Если *новыйтекст* – буфер для перекодированного текста, *новыйкод* – таблица перекодировки (байтовая строка длиной 256, такая, что *новыйкод* [*с*] – новый код символа с кодом *с*), то текст перекодируется и записывается в буфер в результате выполнения операции:

новыйтекст < := *текст* **по** *новыйкод*

Распаковка – разновидность пересылки, которая преобразует цифровой набор в символьное представление числа. Обычно используется совместно с операцией **дес** (п. 1.5.7), преобразующей целое число в цифровой набор. Например, если *номерстроки* – переменная в трансляторе, содержащая номер текущей строки транслируемого текста, то сообщение об ошибке, имеющее вид

строка N: нет закрывающей скобки

формируется следующей операцией пересылки:

сообщение < := "строка" && **распак** (**дес** *номерстроки*)
&& **стр8**": нет закрывающей скобки"

Заметим, что здесь первый источник пересылки – набор, изображающий текст длиной 7 символов. Для распаковки числа со знаком следует вос-

пользоваться тем, что операция дес запоминает знак числа в признаке отношения тго. Последовательность операций:

```
буф [1:] < := распак (дес x);
буф [0] := если тго то" –" иначе " " все
```

записывает в буфер символьное представление числа x со знаком.

Упаковка – операция, обратная распаковке – преобразование числа из символьной формы в цифровой набор. Операция упаковки, как правило, используется совместно с операцией бит (п. 1.5.6), преобразующей цифровой набор в целое число. Если буфер *буф* содержит символьное представление числа со знаком (1 байт – знак, *дл* байтов – число), то операция

```
x := (бит (пак буф [1: дл])). ([63] : буф [0] = "–")
```

присваивает переменной x значение этого целого числа. Необходимое условие выполнения операции пак –*дл* < = 16 (иначе число переводится по частям).

Проверка принадлежности множеству. Операция **среди** проверяет, принадлежит ли начальный символ заданной строки либо заданный символ-набор указанному множеству символов. Например, значение выражения

```
текст среди буквицифр
```

истинно, если символ *текст* [0] – буква или цифра. Выражение

```
" . " среди буквицифр
```

имеет значение **ложь**.

В заключение описания групповых операций над строками приведем решение следующей задачи: выделить из текста целое число без знака и присвоить переменной *число* его числовое значение (целое 64). Если числа нет, либо число превышает максимальное целое формата **ф64**, присвоить *число* **пусто64**.

```
если текст среди цифр
то конст десятьв16 = 10000000000000000,
         максцел64 = цел64 4 " 7 FFFFFFFFFFFFFFF ";
   мод текст от < > "0"; % пропуск нулей
   если текст среди цифр
   то % – – набор значащих цифр – – %
        дл := длбуф;
        буф < :=мод текст длиной мод дл пока среди
                 цифр;
        дл := длбуф – дл; % дл – число цифр
        если дл > =20
        то число := пусто64 % много цифр
        инес % – – перевод числа – – %
              тгз 1; % сброс тго
              дл < =16
        то число := бит (пак буф длиной дл)
```

```
        инес % — — больше 16 цифр; перевод по частям
              число1 := бит (пак буф длиной (дл−16));
              число2:= бит (пак буф [дл−16 : 16]);
            % — — проверка на максцел64 — — %
              число1 > (максцел64−число2) /десятьв16
        то число := пусто64 % большое число
        иначе число:= десятьв16 * число1 + число2
        все
  иначе % кроме нулей, цифр нет; число равно 0
         число:= 0
  все
иначе % цифр в тексте нет
   число:= пусто64
все
```

1.6.7. Область существования и уничтожение векторов. Локальный вектор существует только во время исполнения процедуры или блока, в котором он создан. После завершения или прекращения этого блока он автоматически уничтожается, и все ссылки на него становятся некорректными. Например, после исполнения фрагмента

```
ф64 а;
процедура р = проц ( )
                        (конст в = лок вект [10] ф64;
                        в);
а:= р ( )
```

значением переменной *а* становится указатель уже не существующего вектора *в*, который был уничтожен при завершении процедуры *р*. Попытка использования вектора, например *а* [1], приводит к возникновению ситуации "нет массива".

Глобальный вектор существует при исполнении всей программы. Если глобальный вектор *в* больше не нужен, то его можно уничтожить процедурой ОС

вернутьмассив (*в*)

После вызова этой процедуры попытка обращения к вектору *в* также приведет к ситуации "нет массива".

1.6.8. Косвенная адресация. Любую область памяти (в частности, память переменной) можно рассматривать как вектор. Для этого необходимо сформировать *указатель переменной*, с помощью которого возможен косвенный доступ к этой переменной и изменение ее значения, например:

```
ф64 а, р;
р: = @ а;
р @ : = 1
```

Переменной *р* присваивается указатель переменной *а*, имеющий те же формат и длину, что и сама переменная (формат – **ф64**, длина – 1). Конструкция "косвенная переменная" *р* @ осуществляет косвенный доступ

к переменной a. После присваивания косвенной переменной p @ значение a равно 1. Указатель p можно использовать как обычный указатель: например, p [ф8:] — указатель на область памяти переменной a как на байтовую строку длины 8.

Другой важный способ использования операции "косвенная переменная" — *работа с упакованной информацией.* Дескриптор на область памяти формата **ф1, ф4** или **ф8** и длиной не более слова с помощью операции "косвенная переменная" преобразуется к непосредственно адресуемой *упакованной переменной:*

конст a = **лок вект** [20] **ф8;**

a [3 : 6]@ : = "строка"

После этого присваивания a [3] = "с", a [4] = "т", . . . , a [8] = "а", т.е. a [3: 6] @ ⇔ "строка".

Как видно из примера, упакованная переменная может располагаться в разных словах памяти.

1.7. Многомерные массивы

Для решения прикладных задач в язык Эль-76 введены средства работы с многомерными массивами. По своей мощности и гибкости они превосходят операции над массивами языка Алгол-68, но отличаются от них по форме записи. *Многомерный массив* (далее в п.1.7 слово "многомерный" будем опускать) — это составной объект, доступ к которому осуществляется через *паспорт. Паспорт* содержит *атрибуты* массива, один из которых — указатель *базового вектора,* содержащего элементы массива. Полное название массива (*выстроенный массив,* или *в-массив* [52]) связано с тем, что элементы массива выстроены в линейной области памяти.

Представителем массива в операциях является *указатель паспорта.* Таким образом, доступ к элементам массива осуществляется по следующей схеме:

указатель паспорта → паспорт → базовый вектор → элемент.

Как и векторы, массивы могут быть локальными и глобальными и создаются с помощью генераторов. Однако массивы, в отличие от векторов, не могут иметь переменную длину.

1.7.1. Генератор массива имеет вид

$L\ [l_1, \ldots, l_n]\ F,$

где L — локализация (символ **лок** или **глоб**), n — число измерений массива (n > = 1), $l_1, \ldots, l_n$ — длины по измерениям (выражения), F — формат элементов массива. Формат может быть любым (**ф1, ф4, ф8, ф32, ф64, ф128**). Символ **лок** указывает на создание локального массива, **глоб** — глобального. Результат генератора массива — указатель паспорта. Число измерений n практически неограниченно (в реализации n <= 127).

Использование массива через указатель паспорта возможно с помощью описания константы, например

конст a = **лок** [m, n] **ф64**

или описания переменной

ф64 *а*:= лок [*m*, *n*] **ф64**

Оба описания задают матрицу *а* размера *m* * *n*. Рекомендуется первое описание, так как оно реализуется более эффективно (паспорт размещается в области локальных данных), а также для защиты от ошибок (см. п. 1.6.1).

Особо выделим случай *n* = 1, т.е. случай *одномерного массива.* В этом случае обращение к элементу массива по форме не отличается от обращения к элементу вектора (п. 1.6.3). Поэтому одномерный массив должен описываться с помощью описания константы:

конст *в* = лок [*n*] **ф64**

Если же указатель паспорта одномерного массива является значением переменной или параметра *x*, то с помощью описания константы следует явно указать, что имеется в виду указатель паспорта одномерного массива:

конст *хх* = *пввект* (*x*); ... *хх* [*k*]

Здесь *пввект* (паспорт выстроенного вектора) – стандартная функция. В отличие от вектора одномерный массив адресуется через паспорт; кроме того, если одномерный массив получен в результате операции формирования паспорта (п. 1.7.3), то его элементы могут располагаться в базовом векторе несмежно.

1.7.2. Основные операции над массивами. Операция "элемент массива" записывается обычным образом:

$a\ [i_1, \ldots, i_n]$

Здесь *а* – первичное, значение которого – указатель паспорта *n*-мерного массива; $i_1, \ldots, i_n$ – выражения с целыми значениями. Выход за границу по одному измерению не считается ошибкой. Аппаратно контролируется лишь выход за границу всего массива (т.е. базового вектора). В отличие от векторов при обращении к элементу массива нельзя указывать формат. Нет и операции "подмассив"; выделение подмассива изображается другими средствами (п. 1.7.3).

Атрибуты массива. Массив является составным объектом сложной структуры. В связи с этим основные операции выделения числа измерений, длин по измерениям и указателя базового вектора оформлены как вызовы стандартной функции *читатр* (читать атрибут), предназначенной для выборки атрибутов составных объектов (см. также п. 1.21.3):

читатр (*а*, *члизм*) – число измерений массива *а*;
читатр (*а*, *k*, *длинизм*) – длина по *k*-му измерению;
читатр (*а*, *в*, *длинизм*) – вектор длин по всем измерениям

(*в* – указатель вектора, длина которого равна числу измерений *а*).

Пример.

Пусть описаны две матрицы одинакового размера:

конст *а1* = **лок вект** [*m*, *n*] **ф64**,
а2 = **лок вект** [*m*, *n*] **ф64**

1. Суммирование элементов матрицы *a1*:

```
(ф64 c:= 0;
для i до читатр (a1, 1, длинизм)–1цикл
   для j до читатр (a1, 2, длинизм)–1цикл
        c: = * + a1 [i,j]
   повторить
повторить;
c)
```

2. Пересылка матриц:

```
читатр (a1, базвект) < := ф64 читатр (a2, базвект)
```

В последнем фрагменте операция пересылки матриц, отсутствующая в языке Эль-76, выражена через пересылку их базовых векторов. При выполнении этой операции пересылаются только элементы матриц. В данном случае этого достаточно, так как обе матрицы имеют одинаковые размеры. Однако следует иметь в виду, что для матриц неодинакового размера и для паспортов, полученных операцией **формавм** (п.1.7.3), такая пересылка может дать неправильный результат.

1.7.3. Операции над паспортами. Операция **формавм** (*преобразование формы выстроенного массива*) не имеет аналогов в известных языках программирования. Она, не изменяя содержимого массива и не дублируя его, создает новые паспорта *наложенных массивов*, задающие другую логическую структуру тех же элементов памяти: паспорт транспонированной матрицы, подмассива и т.п. Применение операции рассмотрим на примерах, предполагая, что

конст *мт* = **лок вект** [*n*, *n*] **ф64**

– описание квадратной матрицы.

1) Паспорт транспонированной матрицы

конст *тм* = **формавм** ([*i*, *j*] = *мт* [*j*, *i*])

Здесь результат операции **формавм** – паспорт транспонированной матрицы. Идентификаторы *i* , *j* – *формальные индексы*, их появление слева от знака равенства считается описанием. Идентификаторы *i* и *j* обозначают индекс произвольного элемента наложенного массива *тм* и в правой части используются в выражениях, задающих индексы соответствующего элемента исходного массива. Наложенный массив не может иметь больше измерений, чем исходный. Сложность выражений для индексов в правой части операции ограничена: они могут содержать только линейные комбинации формальных индексов *i*, *j*.

2) Паспорт диагонального вектора матрицы

конст *д* = **формавм** ([*k*] = *мт* [*k*, *k*])

Здесь число формальных индексов равно единице (меньше, чем у исходного массива *мт*).

3) Паспорт побочной диагонали матрицы

конст *пд* = **формавм** ([*k*] = *мт* [*k*, *читатр* (*мт*, 2, *длинизм*)–*k*])

4) Паспорта *i*-й строки и *k*-го столбца матрицы

конст *строка* = **формавм** ([*l*] = *мт* [*i*, *l* – 1]),
        столбец = **формавм** ([*s*] = *мт* [*s*, *k*–1])

5) Паспорт минора (подмассива) матрицы: по первому измерению – от элемента $i1$ длиной $l1$, по второму – от элемента $i2$ длиной $l2$

конст *м* = **формавм** ([*i*, *j*] = *мт* [*i* = *i*1:*l*1, *j* = *i*2: *l*2]),

где $i1{:}l1$ и $i2{:}l2$ – диапазоны изменения индекса i по первому и индекса j – по второму измерениям.

Нижние границы индексов ($i1$, $i2$) матрицы *м* могут быть отличны от нуля. Имеется возможность одновременной установки всех нижних границ в нуль вызовом стандартной функции *запатр* (записать атрибут)

запатр (*м*, *нигрнуль:* **истина**)

После этого индекс i будет изменяться от 0 до $l1 - i1 + 1$, j – от 0 до $l2 - i2 + 1$.

6) Паспорт решетки, составленной из элементов матрицы *мт*, индексы которых по первому измерению четные, по второму – делятся на 3:

конст *р* = **формавм** ([*k*, *l*] = *мт* [*k* * 2, *l* * 3])

Подвижные паспорта. Паспорта массивов, которые рассматривались до сих пор, *постоянны* (описывают один и тот же массив). Возможно также создание *подвижных паспортов*, которые можно накладывать на разные массивы. Например, генератор паспорта

конст *п* = **лок** [,]

создает подвижный паспорт матрицы, настраиваемый на любую матрицу с помощью разновидности операции **формавм**:

формавм (*п* [*i*, *j*] = *мт* [*j*, *i*])

Здесь выполняется присваивание *п* паспорта транспонированной матрицы.

Пример.

Процедура умножения матриц. Для обращения к элементам матриц-сомножителей создаются паспорта их строки и столбца.

```
процедура умнматр = проц (а, в, с)
% умножение матриц а и в. Результат – с
% проверки согласованности для простоты опущены
начало
    конст m = читатр (а, 1, длинизм),
          n = читатр (а, 2, длинизм),
          k = читатр (в, 2, длинизм);
для   i до m – 1 цикл (конст ai = формавм ([s] = а [i, s]);
      для j до k – 1 цикл
            (конст bj = формавм ([s] = в [s, j]);
            ф64 s := 0;
            для t до n – 1 цикл
                  s := * + ai [t] * bj [t]
            повторить;
            c [i, j] := s)
      повторить)
   повторить
конец % – – умнматр – – %
```

В данном примере операция создания паспорта строки и столбца используется лишь в демонстрационных целях. Оптимизация обращения к элементам многомерных массивов в циклах обеспечивается транслятором языка Эль-76 без применения операции **формавм**.

1.8. Особенности структуры программы

Язык Эль-76 имеет более мощные средства структурирования программы, чем известные языки. По структуре программы Эль-76 ближе всего к Алголу-68. По сравнению с последним, Эль-76 не содержит громоздких структурных элементов со сложной семантикой (последовательных предложений с завершителями, совместных предложений, исполняемых описаний с метками), но в нем введен важный класс управляющих конструкций – *структурные предложения.*

Основные отличия структуры программы на Эль-76 от более традиционных языков (Фортран, ПЛ/1, Паскаль) следующие:

1. Эль-76 – *полностью рекурсивный язык:* в нем нет ограничений на сложность операндов в конструкциях. Использование этого свойства придает программе ясность и лаконичность. Например, фрагменту на языке Паскаль

if $x < 1$ **then**

  if $y > 0$ **then** $z := \sin(y)$ **else** $z := \sin(1)$

else if $y > 0$ **then** $z := \cos(y)$ **else** $z = \cos(1)$

эквивалентна гораздо более краткая и наглядная конструкция Эль-76

z := **если** $x < 1$ *то* sin **иначе** cos **все**

  (**если** $y > 0$ **то** y **иначе** 1 **все**)

В данном случае и функция, и аргумент заданы условными предложениями.

2. Эль-76 – *язык выражений:* большинство его конструкций могут использоваться как выражения, т.е. выдают некоторый результат. Этим он отличается от *языка операторов,* в котором четко разграничены выражения (выдающие значение) и операторы (изменяющие состояние вычислений или данных). Языками операторов являются Фортран, Паскаль и многие другие. К числу языков выражений можно отнести из известных языков только Алгол-68, наиболее развитый по структуре программы.

В языке Эль-76 присваивание считается выражением, значение которого равно значению правой части. Поэтому присваивание в скобках можно использовать в выражении. Например, фрагмент программы на языке Паскаль

if $a > b$ **then** $x := a$ **else** $x := b$:

if $c > d$ **then** $y := c$ **else** $y := d$;

if $x > y$ **then** $z := x$ **else** $z := y$;

на языке Эль-76 может быть записан более лаконично:

z := **если** (x := **если** $a > в$ **то** a **иначе** $в$ **все**) >

  (y := **если** $c > d$ **то** c **иначе** d **все**)

  то x **иначе** y

  все

Использование присваивания как выражения рекомендуется только в простых фрагментах, в которых оно не нарушает наглядности программы. Типичный пример:

если ($c := a - в$) > 0 **то** $s1$ **иначе** $s2$ **все**

Здесь использование значения присваивания служит лишь сокращению записи и не повышает эффективность программы, так как эквивалентный фрагмент

$c := a - в$; **если** $c > 0$ **то** $s1$ **иначе** $s2$ **все**

оптимизируется транслятором Эль-76 и транслируется в тот же код, что и первый фрагмент.

Программирование на Эль-76 как на языке выражений возможно и с использованием более сложных конструкций: *замкнутых, условных, выбирающих* и *структурных выражений*, объединяемых понятием *закрытое выражение*. Однако возможен и традиционный "операторный" стиль программирования: многие виды выражений можно использовать и как операторы. Символ ";" после конструкции указывает, что ее результат игнорируется. В некоторых случаях предложение может использоваться только как оператор. Например, условное предложение без альтернативы "иначе" (полный перечень приведен в п. 1.10).

1.9. Описания

Программа на языке Эль-76 состоит из описаний и предложений. Описания входят в состав сложных предложений и, как правило, задаются в их начале (как в языках Алгол-60 и Паскаль и в отличие от языков ПЛ/1 и Алгол-68).

Блоки. Эль-76 – язык с блочной структурой описаний. Блок – это область локализации описаний. Понятие блока в Эль-76, как и в Алголе-68, расширено. Описания разрешены в начале не только последовательного предложения, но и других закрытых предложений:

– в замкнутом предложении (п. 1.11) – после символа "(" или **начало**, действуют до соответствующего символа ")" или слова **конец**;

– в условном предложении (п. 1.12) – от **если** до **все**, от **инес** до **все**, от **иначе** до **все**, от **то** до **иначе**, от **то** до **все** (если альтернатива "**иначе**" отсутствует);

– в выбирающем предложении (п. 1.13) – от **выбор** до **всевыб**, от **иначе** до **всевыб**;

– в цикле (п. 1.14) – от **цикл** до **повторить** (семантика описаний внутри цикла имеет свои особенности).

Описание идентификатора в языке Эль-76 должно предшествовать его использованию.

Неявные описания. Как и в Алголе-68, в Эль-76 введены неявные описания: идентификатор описывается своим появлением в определенной точке программы. Неявно описываются:

– идентификатор счетчика цикла (п. 1.14.1);

– идентификатор статической ситуации в структурном предложении (п. 1.15.1);

– формальные индексы – в операции **формавм** (п. 1.7.3).

Такие неявные описания (в отличие от неявных описаний в Фортране "по первой букве") не снижают надежность программы, так как их появление всегда предсказуемо и ограничено определенным фрагментом текста.

Виды описаний. В зависимости от классов описываемых объектов описания подразделяются на следующие виды:

1. *Описания констант* (п. 1.9.2).

конст *пи* = 3.14159, *дваś10* = 2 ** 10;

2. *Описания переменных* (п. 1.9.1).

ф32 *x*; **ф64** *y* := 1; **ф128** *ee* : = *y*;

3. *Описания процедур* (п. 1.16.1).

процедура *сум* = **функция** (*a*) % суммирование вектора

(**ф64** *c* : = 0;
для *k* **до длина** *a* – 1 цикл
c : = *c* + *a* [*k*]
повторить; *c*) ;

4. *Описания полей* (1.4.3).

поле *знак* = [63], % знак числа
мантисса = [62 :], % мантисса целого 64
знакпор = [62], % знак порядка вещественного 64
порядок = [61: 6], % порядок вещественного 64
мантвещ 64 = [55 :] ; % мантисса вещественного 64

5. *Описания текстов* (1.17).

текст *модуль* (*x*) = *x*. (*знак* : 0), % модуль числа *x*

6. *Описания меток* (1.9.3).

метка *снова, конец*;

7. *Описания статических ситуаций* (1.15.1).

статсит *ошибка, финиш*;

8. *Описание базы* (1.9.4).

база *параметры*;
ф64 *a, в, c*;

Синтаксис описаний. Как видно из примеров, описания начинаются со служебного слова **ф32, ф64, ф128, конст, процедура, поле, текст, метка, статсит**. Хотя порядок описаний произвольный, все же при выборе этого порядка рекомендуется следовать определенной дисциплине (см. ниже). Описания однородных объектов разделяются запятой, разнородных – точкой с запятой.

Порядок описаний. В программе рекомендуется следующий порядок описаний: поля, константы, массивы констант, переменные и массивы, предварительные описания процедур, тексты, непосредственные описания процедур. Такой порядок наиболее логичен. Например, идентификаторы констант могут использоваться для инициализации в описаниях переменных, идентификаторы процедур – в правых частях описаний текстов.

Исполнение описаний. Описания в языке Эль-76 являются исполняемыми конструкциями. Они исполняются в текстуальном поряд-

ке. В примере описаний переменных это свойство используется при описании переменной *ее*.

1.9.1. Описания и использование переменных. Одна из основных особенностей языка Эль-76 состоит в том, что при описании переменной фиксируется не тип, а *формат*. Переменная может иметь только простой формат– **ф32**, **ф64** или **ф128**. Упакованные форматы не допускаются (об упакованных переменных см. п. 1.6.8). В описании переменной может быть задано начальное значение (любое выражение). При использовании переменных необходимо иметь в виду ряд особенностей.

1. Значением переменной может быть значение любого типа, которое помещается в область памяти. Если формат значения меньше формата переменной, то при присваивании оно прижимается к левому краю ее области памяти, а при последующем использовании считывается в исходном формате. Если переменной присваивается число большего формата, то оно преобразуется к формату переменной (1.5.5). В остальных случаях при присваивании значения большего формата возникает прерывание.

2. Для хранения наборов (в частности, логических значений) следует использовать переменные формата **ф64**.

3. При описании без указания начального значения переменная формата **ф32** инициализируется пустым объектом **пусто32**, формата **ф64** – пустым объектом **пусто64**, формата **ф128** – двумя пустыми объектами **пусто64**.

4. В последовательности описаний рекомендуется группировать описания переменных одинакового формата. Например, сначала описать все переменные формата **ф64**, затем – формата **ф32**. Это уменьшит расход памяти в области локальных данных, по сравнению с произвольным чередованием форматов.

Пример.

```
ф32  индекс; % рекомендуется для индексов
ф64  число := 1,
     вектор : = лок вект [10] ф64,
                   % указатель вектора
     совпадение := ложь; % логическая переменная
ф128 хх;           % для чисел двойной точности
индекс := 0;       % индекс ⟨=⟩ цел32 0
число:= цел32 5; % число ⟨=⟩ цел32 5
хх := в128 число;  % значение хх равно вещ128 5
совпадение := число ⟨=⟩ цел32 5; % истина
```

1.9.2. Описания констант служат для мнемонического обозначения постоянных величин идентификаторами, например:

```
конст е = 2.718281828,
      максцел64 = цел64 4 '7FFFFFFFFFFFFFFF",
      начбуф = 1,
      длинабуф = 1024,
      точность = 1е–14;
```

В примере собраны основные случаи использования описаний констант для улучшения наглядности и модифицируемости программ:

– стандартные константы для вычислений (*пи*, *е*, *h*),

— экстремальные значения,

— точность вычислений,

— длины массивов, таблиц и буферов.

Присваивания идентификатору константы запрещены: например, предложение *е*: = 0 ошибочно.

В общем случае в правой части описания констант может быть любое выражение. Если оно вычислимо во время трансляции, то константа в левой части называется *константой статического класса*, а выражение в правой части — *константным выражением*. Например:

конст *квадрант* = *пи* / 2,
максдлвект = 2 * * 20 – 1,
минусодин = . ([63] : 1, [0] : 1)

Описание константы статического класса (числа, набора или пустого объекта) не требует отведения памяти. Если выражение в правой части содержит действия, требующие исполнения, то константу относят к *константам динамического класса*, а ее значение, вычисленное при исполнении, размещается в области локальных данных как значение переменной. В отличие от переменной, для константы динамического класса транслятор обеспечивает защиту от явного присваивания: например, оператор *длтаб* : = 100 ошибочен.

Примеры констант динамического класса:

конст *буфер* = **лок** **вект** [*длинабуф*] **ф64**,
таб16 = **мконст** **ф64** ("0", "1", "2", "3", "4", "5", "6",
"7", "8", "9", "A", "B", "C", "D", "E", "F"),
длтаб = **длина** *таб16*

1.9.3. Описания и использование меток. Метки введены в язык Эль-76 лишь по традиции и для отражения в базовом языке аппаратного типа данных "метка перехода", используемого в основном для реализации различных языков. Средства программирования с использованием ситуаций (п. 1.15) более удобны, наглядны и позволяют обойтись без меток.

Метка, используемая в некотором блоке, должна быть описана в его начале (это описание играет роль предписания):

```
( метка конец;   % описание метки
  ...
  на конец;      % переход
  ...
  конец ⌉: ...   % определяющее вхождение метки
)
```

1.9.4. Описание базы. При исполнении описаний переменных или параметров они размещаются в области локальных данных текущей процедуры. Описание базы позволяет рассматривать часть этой области как вектор. Например, возможно следующее описание параметров и локальных переменных процедуры:

р = **проц** (**ф64** *х*;
база *п*;
ф64 *у*;

```
ф32 z, t )
блок ( % новая область локализации
ф64 пар := n [ф64 : ] ; % пар [0] = y, n[2] = z, n[3] = t
. . .
```

Идентификатор базы *n* обозначает указатель вектора формата **ф32**, описывающий часть области локальных данных от описания базы до ее первого использования (в данном примере – длиной 4 полуслова). Значение переменной *пар* – указатель того же вектора в формате **ф64** (длиной 2). Само описание базы *n* не требует памяти. Описание базы используется для организации процедур с переменным числом параметров (п. 1.16.3).

1.10. Классификация предложений

Предложение – основная структурная единица программы, исполняемая конструкция, которая служит для вычисления значения (*выражение*), имени (*переменная*) или для управления вычислениями (*оператор*). Некоторые выражения могут использоваться как операторы (при этом их значения игнорируются). Частными случаями выражений являются *первичные* – наиболее элементарные конструкции языка Эль-76.

Предложения языка Эль-76 классифицируются следующим образом:

Первичное.	
1. Изображение	1 "авс" **мконст ф64** (1, 2, 3)
2. Идентификатор	*x из16в10*
3. Элемент массива	*вектор* [**ф8** *i*] *матр* [*4 * k* + 1, *4 * k* – 1]
4. Подмассив	*буфер* [*начид* : *длинаид*]
5. Косвенная переменная	*p* @ *числа* [**ф8** 5 : 8] @
6. Вызов *)	*сумма* (*вектор*) *скан* ( )
7. Генератор	**глоб** **вект** [10] **ф64** **лок** [*n*, *n*] **ф32**
	генфайл (*барабан*)
8. Формирование	. (*инф*: 1, *след*: *k*)
	&наб ("эль", "брус")
9. Формирование паспорта *)	**формавм** (*ст1* [*k*] = *матр* [1, *k*]
10. Форматный обмен *)	*запф* (*печ*, **стр8** "конец работы!")
11. Признак	**тгп** **тги** **тго**
12. Подстановка текста *)	*равны* (*x* [*k*], *y* [*k*])
13. Запрос атрибута	*читатр* (*матр*, *базвект*)
14. Модификация атрибута *)	*запатр* (*минор*, *нигрнуль* : **истина**)
15. Поле значения	*x*. *знак* *вектор* [*инд*] . [63 : 8]
16. Закрытое выражение	(**ф64** *а*, *в*; *читф* (*ввод*, **имя** *а*, **имя** *в*) ; *а* + *в*)
17. Внешнее имя	*ксн*//*паскаль*//*новпас* //*код*// [0]
18. Двоичный обмен *)	*зап* (*экран*, *след*, **стр8** "введите *а* :")
Выражение.	
1. Формула	любое первичное
	x > = 0 **и** *x* < = 1
2. Присваивание *)	*вектор* [*k*] : = 0
3. Операция над массивами *)	**мод** *текст* **от не среди** *огран*

О п е р а т о р.

1. Выражение любого вида, помеченного знаком *)
2. Структурный переход — *найден*! (*инд*)
3. Переход — **на** *финиш*
4. Закрытый оператор — **если** *x* **то** *k* := *k* * *k* **все**

```
до решено для n цикл
    c := c + (a := 1/(n * n));
    если a < эпс * c то решено! все
повторить
```

5. Генератор внешнего объекта — *генфайл* (*ув* := *генув* ( ), *ксн*//*паскаль*)
6. Запуск задачи

```
задача (максвремяцп: 1000)
программа
    процедура фиб = проц (печ, k, а, в, с)
        ( если k = 1 то а := 1
          инес k = 2 то в : = 1
          иначе с := а + в; а := в; в := с;
                запф (печ, с)
          все; k:= k + 1); % – – фиб – – %
    процедура р = проц (печ)
        (ф64 k := 1, а, в, с;
        цикл % пока не кончится время
            фиб (печ, имя k, имя а, имя
                в, имя с)
        повторить); + – – р – – %
  р
конец (//вывод)
```

Закрытыми предложениями в языке Эль-76 называются сложные предложения, имеющие открывающую и закрывающую скобки в виде служебного слова или спецсимвола. К ним относятся: *замкнутое предложение* (1.11), *условное предложение* (1.12), *выбирающее предложение* (1.13), *цикл* (1.14), *структурное предложение* (1.15). Закрытые предложения выполняют три функции: используются как выражения (в этом случае они называются *закрытыми выражениями*), как операторы (*закрытые операторы*) и как переменные (*закрытые переменные*). Закрытое предложение может использоваться как выражение (переменная), если во всех его ветвях выдается какое-либо значение (имя переменной).

Переменная – общее понятие, объединяющее все адресуемые объекты языка Эль-76, значения которых можно изменять с помощью операторов присваивания. Классификация переменных:

1. Идентификатор — *а* := *в* + *с*
2. Элемент массива — *вектор* [*k*]: = 1.0
3. Поле переменной — *x. знак* : = 0
4. Косвенная переменная — *р* @ := 1 *буф* [1 : *дл*] @ : = "авс"
5. Закрытая переменная — **если** *x* = 0 **то** *y* **иначе** *x* **все** := 0
6. Подстановка текста — **текст** *зн* (*x*) = *x. знак*; *зн* (*а*) : = 0

В качестве примеров приведены различные формы присваиваний переменным. Переменной простого формата присваивается простое значение, упакованной переменной – содержимое k младших разрядов значения правой части, где k – длина упакованной переменной в битах.

Формулы предназначены для вычисления результатов операций. Порядок исполнения операндов в формуле определяется текстуальным порядком операндов и ***приоритетами операций.*** Операции одного приоритета исполняются слева направо, операции с бо́льшим приоритетом – первыми.

Приоритеты операций (число слева – значение приоритета).

1. **экв** — *число* . *порядок* **экв** 4 "3*F*"
2. **или** — *a* <= *в* **или** *в* <= *a*
3. **и** — *P* **и** *Q*
4. = ⟨ ⟩ ⟨ ⟩ ⟨= ⟩ = ⟨= ⟩ **среди** — *x* ⟨=⟩ **пусто64**
5. + – **плюс** **минус**
6. * **умнд** / /: /: **остат** — *n* * *n* + *m* **остат** *n*
7. ** — 2 ** 20
8. одноместные операции — **естьцел ф64окр** *x*

Пример.
Формула

a = *в* – 1 **или** *c* > *d* + *e* **экв** *x* **среди** *y* **и не** *p*

эквивалентна скобочной формуле:

((*a* = (*в* – 1)) **или** (*c* > (*d* + *e*))) **экв** ((*x* **среди** *y*) **и** (**не** *p*))

1.11. Последовательное и замкнутое предложение

Последовательное предложение в языке Эль-76 состоит из следующих частей:

$D; S_1; \ldots; S_n,$

где D – необязательные описания, S_i – последовательность операторов, R – результат. В последовательном выражении R должно быть выражением, и его значение считается результатом всего предложения. В противном случае последовательное предложение образует оператор.

Описания D действуют до конца последовательного предложения. Замкнутое предложение – это последовательное предложение, заключенное в скобки или в служебные слова **начало** и **конец**. Замкнутое предложение – частный случай первичного; оно может использоваться в любом выражении.

Пример.

(**ф64** *x*; *читф* (*ввод*, **имя** *x*); *x*) +
(**ф64** *y*; *читф* (*ввод*, **имя** *y*); *y*)

Значение этой формулы – сумма двух чисел, введенных из файла *ввод*.

1.12. Условное предложение

Структура условного предложения:

если B_1 **то** P_1
инес B_2 **то** P_2
. . .
инес B_n **то** P_n
иначе P
все

Здесь B_i – условия; P_i, P – последовательные предложения. Условие B_i может быть задано выражением: D; S_1; . . . ; S_k; B. В последнем случае его описания D действуют до конца условного предложения (до слова **все**); $S_1, \ldots, S_k$ – операторы, B – выражение. Альтернативы **инес** и **иначе** могут отсутствовать. Использование условного предложения без альтернативы **иначе** в качестве выражения не допускается.

Особенность семантики условного предложения языка Эль-76 в том, что условие может быть представлено значением любого типа B и эквивалентно условию B . [0] (т.е. учитывается только младший бит B). Поэтому при использовании условных предложений следует соблюдать осторожность. В практике программирования на Эль-76 встречаются ошибки, когда условие оказывается представленным пустым объектом, меткой процедуры и т.п., что может привести к недетерминированному поведению программы.

Примеры.

1. Условное выражение:

если $a > b$ **то** a
иначе b
все

2. Условный оператор (не может быть использован в качестве условного выражения):

если *число* % нечетно
то *степень* := *степень* * *множитель*
все;

3. Неправильное использование условных предложений (динамическая ошибка не фиксируется):

а) **если ф64** *a*; *a* % значение *a* не определено (**пусто64**)
то *S*1
иначе *S*2
все % будет исполняться всегда *S*2

б) **процедура** *p* = **функция** ( ) (. . .);
если *p* % пропущены скобки в вызове: *p* ( )
то *S*1
иначе *S*2
все % недетерминизм: будет исполняться *S*1 либо *S*2, в
% зависимости от 0-го бита адреса в метке процедуры *p*

1.13. Выбирающее предложение

Выбирающее предложение языка Эль-76 – аналог оператора **case** в языке Паскаль и предложения **case** в Алголе-68. Осуществляет выбор для исполнения одного из последовательных предложений $S_1, \ldots, S_k$ в зависимости от *номера* n – целого числа или набора. Вычисление номера n записывается так же, как вычисление условия в условном предложении (п. 1.12). Структура выбирающего предложения:

выбор n **из**
  $c_1 : S_1,$
  ...
  $c_k : S_k$
иначе S
всевыб

Здесь n – вычисление номера; c_i – константные выражения, задающие возможные значения n (*разметка*); S_i, S – последовательные предложения. Исполнение выбирающего предложения приводит к исполнению S_i, если $c_i = n$ (константы c_i не должны повторяться), либо к исполнению S, если среди c_i нет значения n. Альтернатива **иначе** может отсутствовать, но в этом случае выбирающее предложение не может использоваться как выражение. Если предложение S должно исполняться при нескольких значениях n (например, c_1, c_2 и c_3), то соответствующая альтернатива выбирающего предложения записывается как c_1:, c_2:, $c_3 : S$. Константы c_i (все одновременно) можно опускать вместе с двоеточиями; по умолчанию они равны 0, 1, 2, ...

Если выбирающее предложение используется как оператор, а по смыслу программы альтернативы **иначе** не требуется (перечень констант выбора является исчерпывающим), то для контроля возможных ошибок рекомендуется ввести альтернативу **иначе** с сообщением об ошибке, например:

... **иначе** *запф* (*печ*, стр8 "недопустимое значение n = ", n)
всевыб

Примеры.

1. **выбор** *число* **из**
    "нуль", "один", "два", "три"
  иначе "много"
  всевыб
2. **выбор тип** x **из**
    цел32 :, **цел64** : *печцел* (x),
    вещ32 :, **вещ64** :, **вещ128** : *печвещ* (x),
    деск : *печмассив* (x)
  иначе *печшестн* (x)
  всевыб

Второй пример демонстрирует выбор по типу значения x; выбирающее предложение в примере 2 аналогично специальным выбирающим предложениям языков Алгол-68 и Клу для обработки значений объединенных видов (**union**, **oneof**, **variant**). С точки зрения, принятой в этих языках, любая

переменная динамического типа в языке Эль-76 имеет объединенный тип, равный объединению всех, в том числе определяемых пользователем, типов. В расширенном языке Эль-76 [52] для анализа типа объекта введен следующий класс управляющих конструкций – *выбирающее по типу.*

3. Ошибочное выбирающее выражение:

```
a := выбор в. [2 : 3] из
        "0", "1", "2", "3", "4", "5", "6", "7"
    всевыб
```

Несмотря на то, что значение номера по смыслу программы всегда лежит в интервале от 0 до 7, данное выбирающее предложение не является выражением, так как в нем отсутствует альтернатива **иначе**. Эта ошибка диагностируется транслятором.

1.14. Циклы

Как и большинство языков, язык Эль-76 имеет два вида циклов: с известным числом итераций – *цикл с заголовком* и с неизвестным числом итераций – *простой цикл*. Циклы не вычисляют никакого явного результата и могут использоваться только как операторы.

1.14.1. Цикл с заголовком имеет вид:

для i **от** m **до** n **цикл**
$D; S_1; \ldots; S_n$
повторить

Здесь i – *идентификатор счетчика цикла* (это вхождение i считается его описанием); m, n – границы изменения i (выражения); D – необязательные локальные описания; S_i – операторы. Значения m и n вычисляются один раз, перед началом повторений цикла. Шаг цикла может быть равен 1 или –1 (шаг –1 задается служебным словом **вниздо** вместо слова **до**). Описания D исполняются один раз, перед началом итераций. Умолчания:

1. **от** 0
2. **до** 2 * * 20–2
3. если опущено **для** i, то создается внутренний счетчик цикла, недоступный программе. В этом случае заголовок цикла должен начинаться со слова **от**, чтобы цикл можно было отличить от структурного предложения (п. 1.15).

П р и м е р.

Сортировка методом пузырька

```
для i до n – 2 цикл
      ф64    imin, x;
        imin := i;
      для    j от i + 1 до n – 1
      цикл
             если a[imin] > a[j]
             то imin := j
             все
```

```
    повторить;
        x := a[i];
        a[i] := a[imin];
        a [imin] := x
повторить
```

Отметим здесь особенность инициализации локальной переменной внешнего цикла *imin*, описание которой исполняется один раз. Совмещение инициализации с описанием (**ф64** *imin* := *i*) привело бы к неверному результату: переменной *imin* было бы присвоено *начальное* значение *i*, т.е. 0, в то время как по смыслу программы требуется, чтобы присваивание переменной выполнялось *на каждом шаге* цикла при новом значении *i*.

В некоторых случаях, однако, более удобно, чтобы локальные описания в цикле исполнялись каждый раз заново, при новом значении параметра цикла. Для этого тело цикла следует заключить в дополнительные скобки; полученное замкнутое предложение образует новую область локализации (см. пример процедуры **умнматр** в п. 1.7.3). Другая форма записи предыдущего примера:

```
для i до n – 2 цикл
    (ф64 imin := i; . . . % далее как в предыдущем примере
) повторить
```

Описание переменной *imin* и ее инициализация очередным значением *i* исполняются на каждом шаге цикла.

Как и в Алголе-68, параметр цикла *i* в теле цикла, на каждом его шаге, считается константой. Как и обычные константы, его нельзя изменять с помощью присваивания: *i* := *e*; нельзя также создавать на него ссылку, с помощью которой его значение можно было бы изменить, например, передавать в процедуру по имени: *p* (**имя** *i*) или создавать на него указатель: @*i*.

В отличие от языков Паскаль и Алгол-60 в языке Эль-76 при завершении цикла, а также при выходе из него по метке или по ситуации, идентификатор параметра цикла теряет смысл. Если же требуется сохранить его текущее значение после выхода из цикла, то следует присвоить его глобальной переменной или передать как параметр структурного перехода (1.15). Рекомендуется второй способ ввиду большей наглядности.

П р и м е р.

Суммирование числовых элементов массива.

```
до пустой, нечисло
 для i  до длина a – 1 цикл
        если естьпусто a [i]  то пустой! (i)
        инес естьцел a [i]  то s : = s + a [i]
        иначе нечисло! (i)
        все
при пустой (k): запф (печ, стр8 "пустой элемент", k)
        нечисло (k): запф (печ, стр8 "не числовой элемент", k)
всесит
```

1.14.2. **Простой цикл** записывается в следующей форме:

цикл $D; S_1; \ldots; S_n$ **повторить**

где D, S_i имеют тот же смысл, что и в п. 1.14.1. Простой цикл исполняет-

ся бесконечно, т.е. до прекращения с помощью перехода (п. 1.9.3) или структурного перехода (п. 1.15). Как правило, простой цикл используется совместно с объемлющим его структурным предложением (п. 1.15.1).

Пример. Метод итераций:

```
цикл
        ф64     u := x;
        если    x := f(u);
                равны (x, u)
        то на   конец
        иначе   u := x
        все
повторить
```

1.15. Структурное предложение

Структурное предложение – конструкция языка Эль-76, предназначенная для обработки *ситуаций*. Понятие ситуации складывалось в языках программирования постепенно, начиная с языка ПЛ/1. Некоторые широко распространенные языки (Паскаль, Модула-2) средств обработки ситуаций не имеют. В более развитой форме механизм ситуаций был включен в языки Клу и Ада. Однако впервые ситуации в их наиболее полном современном понимании были введены в 1973 г., в первой версии языка Эль-76 [52].

Понятие ситуации объединяет следующие классы особых условий завершения конструкций.

1. Условия завершения сложного предложения, предусмотренные программистом: достижение нужной точности при вычислениях, обнаружение или отсутствие элемента в структуре данных, успешный или неудачный поиск варианта решения и т.п. Ввиду непосредственной связи ситуаций этого класса со статической структурой текста программы они называются *статическими ситуациями*.

2. Особые условия завершения процедуры, рассматриваемые как часть ее интерфейса (видимого эффекта): например, ситуация "пуст" для функции "голова списка"; ситуация "логический конец файла" для процедуры буферизованного обмена. Принципиальное отличие от статических ситуаций – отсутствие фиксированной связи с определенной точкой в тексте программы; динамическая связь места возникновения ситуации и места ее обработки, возникающая при вызове процедуры. Поэтому ситуации данного класса называются *динамическими*.

3. Прерывания при выполнении стандартных операций (переполнение, деление на нуль, ошибки адресации и т.д.). Ситуации этого класса называются *стандартными*. По характеру связи с местом обработки ситуации их можно причислить к динамическим, однако их специфика в том, что они приводят к прерываниям процессора и вызову операционной системы.

Структурное предложение языка Эль-76 – единая конструкция для обработки всех трех классов ситуаций: статических, динамических и стандартных. Однако обработка статических ситуаций более проста – как при программировании, так и в реализации.

1.15.1. Статические ситуации. Рассмотрим примеры наиболее распространенных случаев использования структурных предложений для обработки статических ситуаций. Особенности синтаксиса и семантики Эль-76 разъясняются на этих примерах.

1. Общий алгоритм поиска элемента в списке. Список задается ссылкой на первый элемент *нач*, функциями выделения информационной части *инф* (x) и ссылки на следующий элемент *след* (x). Конец списка – нулевая ссылка.

При обработке списка возможны две статические ситуации: *найден* и *конецсписка*.

```
до найден, конецсписка
цикл
        ф64 тек : = нач; % ссылка на текущий элемент
        если тек = 0
        то конецсписка!
        инес инф (тек) = x
        то найден! (тек)
        иначе  % переход к следующему элементу
               тек : = след (тек)
        все
повторить
при найден (инд): обработать (инд),
        конецсписка: запф (печ, стр8 "элемент не найден")
всесит
```

Структурное предложение начинается с *заголовка структурного* – слова **до** и списка идентификаторов ситуаций (их вхождения считаются описаниями). Затем помещается цикл, управляемый структурным предложением (вместо цикла может быть любое другое закрытое предложение). Для прекращения цикла выполняется *структурный переход* по соответствующей ситуации, например, *конецсписка*!. В результате исполняется *реакция* на ситуацию, помеченная ее именем (в примере – печатается сообщение), и структурное предложение завершается. Реакции объединяются в *приставку действий*, заключенную между словами **при** и **всесит**. Структурный переход по ситуации *найден* выполняется с параметром *тек*; соответствующая реакция имеет формальный параметр. Параметры структурного перехода служат для передачи локальных результатов вычислений из вложенного закрытого предложения.

Применение структурных предложений значительно повышает наглядность программы. Программирование тех же действий с помощью меток приводит к чрезмерному количеству переходов и к нарушению явной связи текста с логической структурой алгоритма. Структурные предложения не ухудшают эффективность программы, так как статические ситуации реализуются с помощью обычных команд переходов (как и переходы по метке).

2. Приближенное вычисление значения функции

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

до *готово*

```
для k от 1 цикл
        конст eps = 1e − 15;   % точность
        ф64    a := 1,         % общий член ряда
               s := 1;         % сумма ряда
        если
               a := a * x/k;
               s := s + a;
               модуль (a) < eps * модуль (s)
        то готово! (s)
        все
повторить
```

В этом примере структурное предложение имеет следующие особенности:

– определена одна ситуация *готово*; реакция на нее не задана. Если реакции на все ситуации отсутствуют, то приставка действий опускается (концом структурного предложения в данном случае является конец цикла, т.е. символ **повторить**);

– структурный переход по ситуации *готово* выполняется с одним параметром *s*. Поскольку реакция не задана, значение параметра считается *результатом* структурного предложения. Таким образом, оно является *структурным выражением,* хотя управляемый им цикл не выдает никакого результата.

3. Организация прекращения работы программы при выполнении любой ее процедуры.

```
начало % программы
   статсит авария;
   процедура итерация = проц ( )
               ( ... авария! (пусто64) ... );
   ...
   процедура решение = проц ( )
       (ф64 числоитераций := 1;
       ... итерация ( );
       если числоитераций > 1000
       то авария! (числоитераций)
       все ... ); % конец "решение"
% − − основная часть − − − %
до * авария
   ( ... решение ( ) ... )
при    авария (x) :
       если естьцел x
       то запф (печ, стр8 "превышено число итераций")
       иначе запф (печ, стр8 "ошибка в исходных данных")
       все
       всесит
конец
```

В отличие от двух предыдущих примеров, в примере 3 статическая ситуация *авария* может возникнуть в нескольких точках программы (в процедуре

итерация или в процедуре *решение*), а место обработки этой ситуации (основная часть программы) текстуально отделено от места возникновения ситуации. Для предварительного описания идентификатора ситуации в таких случаях используется *описание статической ситуации* (статсит *авария*). В структурном предложении, в котором выполняется обработка этой ситуации, перед идентификатором ситуации *авария* ставится звездочка для отличия *упоминания* предварительно описанной *ситуации авария* от нового описания ситуации с тем же именем. Следует иметь в виду, что если звездочка ошибочно пропущена (весьма распространенная ошибка), то фрагмент до *авария* будет воспринят как описание *новой* ситуации. При этом ситуация *авария*, возникающая в процедурах *решение* и *итерация*, обработана не будет. Такая ошибка (необработанная статическая ситуация) диагностируется транслятором.

Статические ситуации с предописаниями рекомендуется, как и в данном примере, использовать для организации завершения или прекращения программы. Таким образом, и в этой роли статические ситуации успешно заменяют метки, а дополнительным преимуществом ситуаций является возможность передачи параметров при структурном переходе.

На использование статических ситуаций накладывается следующее ограничение. Если в программе имеются *два* структурных предложения для обработки *одной* статической ситуации, то исполнение одного из них не должно привести к исполнению другого. Например, в процедуре *решение* или в вызываемых из нее процедурах не может быть задано еще одно структурное предложение для обработки ситуации *авария*.

1.15.2. Динамические ситуации. Для более гибкого управления вычислениями может оказаться недостаточной схема использования статической ситуации:

определение s → *s*! → *обработка s*

А именно, может потребоваться *переопределение реакции* на ситуацию *s* при исполнении программы, т.е. в последовательности вызванных процедур. В примере 3 из п. 1.15.1 представляется удобным определить в процедуре *решение* собственную реакцию на ситуацию *авария*, возникающую в процедуре *итерация*, так как это позволяет не прерывать весь процесс решения.

Другой характерный пример – из области системного программирования: организация интерпретатора языка с блочной структурой. Наиболее удобна процедурная организация интерпретатора: каждой конструкции (например, блоку) соответствует своя процедура интерпретации (*блок*). При интерпретации *статическая вложенность* блоков в исходном тексте программы на интерпретируемом языке переходит в *динамическую вложенность* экземпляров (т.е. последовательность рекурсивных вызовов) процедуры *блок*. Возникает проблема интерпретации перехода по метке из одного блока в другой, которая сводится к следующей задаче: организовать выход из *n* экземпляров рекурсивной процедуры *блок* (где *n* определяется статически или при интерпретации).

Для решения подобных задач управления программой в язык Эль-76 введены *динамические ситуации*. Основное отличие их от статических в том, что они распространяются не по тексту программы, а по ее *динами-*

ческой цепочке – цепочке вызванных процедур, в направлении, обратном порядку вызовов.

Создание указателя ситуации. Для создания указателя динамической ситуации служит *генератор ситуации*, который в большинстве случаев имеет вид:

```
ситуация ( )
```

Описание динамической ситуации, как и описание массива, строится как описание константы или переменной:

```
конст авария = ситуация ( )
```

Структурный переход и обработка (решение первой задачи). Если в примере 3 п. 1.15.1 описание статической ситуации заменить указанным описанием динамической ситуации, то структурное предложение и структурные переходы не требуют каких-либо изменений, а в процедуре *решение* можно будет переопределить реакцию на динамическую ситуацию *авария*:

```
процедура решение = проц ( )
   (ф64  числоитераций := 1;
   до     * авария
          (...итерация ( ) ...)
   при авария: реакция ( )
   всесит;
   если числоитераций > 1000
   то авария! (числоитераций)
   все...); %– – – решение – – –%
```

При возникновении ситуации *авария* в процедуре *итерация* обработка ситуации будет выполнена в процедуре *решение* с помощью процедуры *реакция* ( ), после чего исполнение процедуры *решение* будет продолжено (будет исполняться условное предложение, следующее за структурным предложением, которое обрабатывает ситуацию *авария*). Если же ситуация *авария* возникает в процедуре *решение*, то процедура *решение* будет прекращена и будет запущена реакция в основной части программы.

Упоминание динамической ситуации, структурный переход и обработка динамической ситуации по форме не отличаются от конструкций для обработки статической ситуации с предописанием.

Использование динамических ситуаций для управления рекурсивными процедурами (решение второй задачи). Для реализации оператора перехода в интерпретаторе введем два глобальных описания:

```
конст переход = ситуация ( );
ф64 уровеньзапуска : = 0
```

Переменная *уровеньзапуска* содержит текущее число запущенных, но не завершенных блоков. Метке, помечающей оператор в блоке, поставим в соответствие структуру *метка* из двух полей: *уровень* – уровень запуска экземпляра блока, где описана метка, и *номер* – номер оператора в этом блоке. Приведем реализацию процедур *блок* (интерпретация блока) и

операторперехода (интерпретация оператора перехода).

```
процедура блок = проц (чисопер) % число операторов блока
начало % — — — блок — — — %
  ф64 номопер := 0, % текущий номер оператора
      урблока := уровеньзапуска;
                              % уровень запуска текущего блока
  % — — — тело процедуры блок — — — %
  уровеньзапуска := * + 1;
  до конец цикл
     до * переход цикл
        если номопер >= чисопер
        то конец! % исполнение блока завершено
        иначе оператор ( ); % очередной оператор блока
              номопер := * + 1
        все
     повторить
     при переход (ном, ур):
     если ур = урблока
     то % — — переход достиг нужного блока
        уровеньзапуска := ур;
        номопер := ном % переход к оператору номер ном
     иначе % — — продолжение перехода
        переход! (ном, ур-1)
     все
  всесит
  повторить ; % — — до следующего перехода или до конца
  уровеньзапуска := * — 1
  конец ; % — — — блок — — — %
процедура операторперехода = проц (метка)
       (переход! (метка . номер, метка . уровень));
```

В примере подразумевается, что процедуру *операторперехода* запускает процедура *оператор* при интерпретации оператора перехода. Значение метки берется из глобальных величин блока. Оператор перехода реализуется с помощью структурного перехода по динамической ситуации *переход*. В процедуре *блок* внешний цикл используется для первоначального запуска блока и его повторного запуска после очередного перехода в данный блок. Статическая ситуация *конец* соответствует завершению блока. Внутренний цикл последовательно интерпретирует операторы блока до очередного перехода. Переход приводит к прекращению процедуры *оператор* и запуску реакции. В реакции проверяется уровень запуска блока. Если нужный уровень (т.е. блок) достигнут, то внешний цикл возобновляет исполнение блока начиная с указанного номера оператора. В противном случае вновь выполняется структурный переход по той же ситуации с меньшим на единицу номером уровня. Он приводит к прекращению текущего поколения процедуры *блок* (текущего блока) и будет обработан реакцией в предыдущем поколении процедуры *блок* (т.е. в объемлющем блоке) и т.д.

В данном примере использованы все возможности динамических ситуаций: переход по динамической цепочке, переопределение реакции, пере-

ход из реакции на ситуацию в реакцию на ту же ситуацию в вызывающей процедуре. Для статических ситуаций ни одна из этих возможностей не имеет места.

Атрибуты динамической ситуации. В генераторе ситуации можно задавать значения двух атрибутов: *процкр* (процедура конечной реакции) и *локал* (признак локализации в текущем блоке). Например:

ситуация (*процкр*: *p*, *локал*: **ложь**)

По умолчанию ситуация считается локальной, а конечная реакция не предусмотрена. Процедура *p* конечной реакции запускается, если ситуация не обрабатывается, т.е. в программе (более точно – в независимом процессе, п. 1.30.2) не найдена соответствующая реакция. Процедура *p* может содержать заключительные действия по выдаче сообщений и т.п. После запуска конечной реакции (и при ее отсутствии) процесс уничтожается.

1.15.3. Стандартные ситуации. Любая динамическая ошибка в языке Эль-76 (прерывание, ошибка при работе с файлами и т.п.) рассматривается как стандартная динамическая ситуация и, следовательно, может быть обработана структурным предложением. Каждая стандартная ситуация имеет свой предопределенный идентификатор и связанный с ним объект типа "указатель динамической ситуации". Имеются следующие стандартные ситуации:

границамассива – выход за границу массива при индексации (п. 1.6.7);

делениенануль – деление на нуль в арифметической операции (п. 1.5.2);

исчерпвремяопрц – большое время выполнения операции, например зацикливание в операции поиска по списку (п. 1.6.4);

нарушениезащиты – выполнение действий, запрещенных защитой, например попытка изменения содержимого массива констант (п. 1.6.5);

неверныйоперанд – недопустимый тип или значение операнда в операции;

недопустимыйаргумент – недопустимый аргумент в стандартной функции (tg, ln и т.п.);

ошибкаимени – обращение по несуществующему имени (например, из-за недостаточного числа фактических параметров процедуры) или в середину значения (например, попытка считывания из младшей половины слова, в котором хранится целое 64);

переполнвещ128, *переполнвещ*, *переполнцел32*, *переполнцел64* – переполнение в операциях над числовыми значениями соответствующих типов и форматов;

привдействия – привилегированные действия в непривилегированном режиме (например, попытка создания указателя вектора с помощью операции "формирование", п. 1.4.3);

ситнетмас – обращение по несуществующему адресу (например, использование локального массива после выхода из блока, в котором он локализован, п. 1.6.7).

Большая группа ситуаций возникает при работе с внешними объектами. Эти ситуации описаны в п. 1.28.

Любую стандартную ситуацию программист может использовать по своему усмотрению, если это соответствует логике программы. Например,

при реализации процедуры умножения матриц можно использовать структурный переход *неверныйоперанд*! в случае, если размеры матриц не согласованы.

Примеры.

1. Модификация процедуры скалярного умножения векторов, учитывающая возможные динамические ошибки.

```
процедура скал = функция (x, y)
   до  * неверныйоперанд, * границамассива
      (ф64 c := 0;
      для k до длина x – 1
      цикл
            c := * + x [k] * y [k]
      повторить; c)
   при  неверныйоперанд:, границамассива:
      пусто64
   всесит
```

Если либо x, либо y – не векторы или какой-либо элемент одного из векторов – не число, то возникает ситуация *неверный операнд*. Если длины векторов не равны, может возникнуть ситуация *границамассива*. В этих случаях результатом процедуры является значение **пусто64**.

При программировании подобных процедур, как и процедур с предописываемыми статическими ситуациями (п. 1.15.1), необходимо тщательно проверять расстановку звездочек в заголовках структурных предложений: заголовок **до** *неверныйоперанд, границамассива* означает описание двух *новых* статических ситуаций, а стандартные ситуации с этими идентификаторами обработаны не будут.

2. Использование стандартной ситуации при программировании:

```
процедура sqrt = функция (x)
   если x < 0
   то неверныйоперанд!
   иначе  % вычисление корня
            . . .
   все
```

1.16. Процедуры

Понятие процедуры играет фундаментальную роль в языке Эль-76, как и в других языках программирования. Отличие процедурного механизма Эль-76 от традиционного – в значительно большей степени динамизма. Так, процедура может выступать как объект типа **мпроц** (метка процедуры), т.е. может присваиваться, передаваться параметром; для процедур стандартного (динамического) типа не контролируется соответствие числа и типов параметров при вызове процедуры, что дает возможность написания процедур с переменным числом параметров.

Все элементы процедурного механизма Эль-76 имеют полную аппаратную поддержку, что обеспечивает его эффективность.

1.16.1. Описание и использование процедур. Для изображения процедуры служит конструкция "текст процедуры", близкая по форме аналогич-

ной конструкции Алгола-68. Например:

```
функция (x, y) % возведение в натуральную степень y
   (ф64 множитель := x,
         степень := 1;
   для позиция до перв1 y
   цикл
      если y. [позиция]
      то степень := степень * множитель
      все;
      множитель := множитель * множитель
   повторить;
  степень)
```

Слово **функция** – признак наличия *результата* (если результата нет или он игнорируется, используется слово **проц**). В заголовке процедуры указывается *список* ее *формальных параметров*. Если их нет, пустые скобки обязательны. *Описание формальных параметров* имеют вид описаний переменных без инициализации (их начальные значения задаются при вызове процедуры). Как правило, параметры имеют формат **ф64**, поэтому он подразумевается по умолчанию (*x*, *y*) эквивалентно (**ф64** *x*, *y*). *Тело* процедуры – *закрытое предложение* (в данном примере – замкнутое предложение). *Значение* конструкции "текст процедуры" – *метка*, которая используется для вызова процедуры.

Описание процедуры связывает идентификатор с ее текстом:

```
процедура натстепень = функция (x, y) (...)
```

Процедуру-функцию *натстепень* можно использовать в *выражениях*, например: *натстепень* (*натстепень* (*a*, *в*), *c*). Процедуры со спецификацией **проц** используются как *операторы*:

```
процедура трансп = проц (a) % транспонирование матрицы
  для i до читатр (a, 1, длинизм) – 1
  цикл ф64 x;
        для j до i – 1 цикл
            x := a [i, j];
            a [i, j] := a [j, i];
            a [j, i] := x
        повторить .
  повторить; % – – – трансп – – – %
трансп (м1);
```

Для описания взаимно рекурсивных процедур введены *предварительные описания процедур* (напомним, что в языке Эль-76 описание идентификатора должно предшествовать его использованию):

```
процедура терм; % предварительное описание
процедура
  выражение = проц ( ) (... терм ( ) ...),
  терм = проц ( ) (... выражение ( ) ...)
```

В больших программах рекомендуется предварительно описывать все процедуры. Это позволяет располагать их *непосредственные* (окончательные) *описания* в произвольном, удобном для программиста порядке.

1.16.2. Вызов процедуры и передача параметров. Вызов процедуры без параметров в языке Эль-76 имеет вид

$p(\ )$

(скобки обязательны!). Вызов процедуры с параметрами записывается в форме:

$p(x1, \ldots, xk)$

Здесь p – первичное, обозначающее процедуру; $x1, \ldots, xk$ – фактические параметры. Вызов процедуры имеет следующие особенности.

1. В языке Эль-76, в отличие от большинства языков, способ передачи параметра указывается не в описании процедуры, а при вызове: каждому из параметров x_i может предшествовать *спецификация способа передачи.* Уровень системы команд МВК "Эльбрус" позволяет оттранслировать описание процедуры без использования информации о способах передачи параметров.

2. При вызове процедуры стандартного (динамического) типа не контролируется соответствие числа и типов параметров. Поэтому передача неверного числа параметров может привести к смещению области локальных данных, что, в свою очередь, приводит либо к неверной работе процедуры, либо к стандартной ситуации *ошибкаимени* (п. 1.16.3).

3. При программировании следует тщательно проверять наличие скобок в вызовах процедур без параметров. В силу особенностей Эль-76 выражение p (где p – процедура) обозначает *саму* эту *процедуру,* т.е. ее *метку.* Вызов процедуры без параметров осуществляется с помощью конструкции $p(\ )$. Следовательно, пропуск скобок в вызове приводит к тому, что процедура *не запускается.* Как показывает опыт программирования на Эль-76, даже для высококвалифицированных программистов эта ошибка – одна из самых распространенных и трудно обнаруживаемых. Для ее поиска требуется организация трассировки работы процедур.

1. Передача значением осуществляется по умолчанию (без явной спецификации): например, $\sin(x + 1)$. Фактический параметр должен быть *выражением.*

При передаче значением необходимо соблюдать соответствие форматов фактических параметров описанию процедуры. Как правило, формальные параметры имеют формат **ф64**.

Параметр-значение можно рассматривать и использовать как обычную локальную переменную, начальное значение которой задается при вызове процедуры.

2. Передача именем специфицируется словом **имя**. Фактический параметр должен быть *переменной* (п. 1.10). Вычисляется и передается ее *адрес.* Иначе говоря, этот способ соответствует традиционному способу передачи *по ссылке,* принятому в языках Фортран-IV и ПЛ/I, и передаче *параметра-переменной* (var) в языке Паскаль. В теле процедуры использование значения такого параметра x приводит к использованию значения фактической переменной v, а присваивание x приводит к присваиванию v.

П р и м е р.

процедура *обменять* = **проц** (x, y)

```
   % -- обменять местами значения двух переменных -- %
   (ф64  a := x;
         x := y
         y := a); % -- обменять -- %
```

Последовательность операторов

$a := 1$; $в := 2$; *обменять* (**имя** a, **имя** $в$)

приводит к тому, что $a = 2$, $в = 1$; последовательность операторов

$a[i] := 0$; $a[j] := 1$; *обменять* (**имя** $a[i]$, **имя** $a[j]$)

к тому, что $a[i] = 1$, $a[j] = 0$. Если в процедуру *обменять* (после описания переменной a) добавить присваивание $j := 1$, то эффект процедуры не изменится, так как адрес переменной вычисляется в момент передачи.

3. Передача значения подпрограммой (спецификация **прог**). Этот способ соответствует *вызову параметра по наименованию* в языке Алгол-60 (отметим, что в более поздних языках он практически не применялся из-за сложности реализации). При этом способе передается *подпрограмма* (*техническая процедура*, см. п. 6.1.1), обеспечивающая вычисление параметра-выражения при каждом его использовании в теле процедуры.

П р и м е р.

Процедура обобщенного суммирования, которую можно использовать как знак суммы $\sum_{i=m}^{n} a_i$ [42].

```
процедура сумма = функция (k, m, n, a)
   (ф64  c := 0;
   k := m;
   от  m до n цикл
        c := c + a;  k := k + 1
   повторить;
   c) % --- сумма --- %
```

Использование:

запф (*печ*, *сумма* (**имя** j, 1, 20, **прог** $1/j$ ** 2))

Печатается значение суммы $\sum_{j=1}^{20} \frac{1}{j^2}$. Здесь j – некоторая переменная.

В теле процедуры последовательно изменяется значение переменной j (от 1 до 20), поэтому использование параметра a на каждом шаге ведет к вычислению выражения $1/j$ ** 2 с новым значением j.

4. Передача имени подпрограммой (спецификация – **имя прог**). Отличается от **прог** тем, что параметр должен быть *переменной*, т.е. конструкцией, вычисляющей **имя**.

Пример.

Процедура обобщенного присваивания.

```
процедура присв = проц (k, m, n, a, в)
   (k := m;
        от m до n цикл
           a := в;
           k: = k + 1
        повторить ) % -- присв -- %
```

Использование:

```
присв (имя j, 0, длина a – 1, имя прог a[j],
        прог в [j] + c [j])
```

– суммирование векторов;

```
присв (имя j, 0, длина s – 1, имя прог s[j],
прог s1 [j] или s2[j])
```

– объединение множеств.

Процедурные параметры. Передача параметров-процедур и параметров-функций в языке Эль-76 является частным случаем передачи значением (в качестве значения выступает метка процедуры).

Пример.

Вычисления корня функции методом дихотомии.

```
процедура корень = функция (f, a, в)
% f непрерывна; a < в; f(a) * f(b) < 0
до вычислено цикл
    конст eps = 1e – 14;
    ф64 m, x = a, y = в, fx := f(a), fy := f(b), fm;
    если y – x < eps то вычислено! (x)
    инес fm := f(m := 0.5 * (a + в));
      fx . [63] <>fm [63] % f(x) и f(m) разных знаков
    то  y := m; fy := fm
    иначе x := m; fx := fm
    все
повторить % --- корень --- %
```

Использование:

```
запф (печ, корень (функция (x) (x – 1), 0, 2))
```

Здесь функция f задана явно – текстом процедуры.

1.16.3. Процедуры с переменным числом параметров. Для организации процедур с переменным числом параметров используется прием, позволяющий рассматривать область фактических параметров как вектор. Для этой цели служит конструкция "описание базы". Единственная проблема состоит в том, что локальные переменные процедуры размещаются в той же области, что и фактические параметры. Для отделения области локальных данных от области фактических параметров применяется конструкция **блок** s, где s – закрытое предложение тела процедуры. Локальные величины предложения s размещаются в новой области локальных данных, т.е. конструкция **блок** s транслируется как конструкция **проц** ( ) s ( ).

П р и м е р.

Процедура вычисления минимума произвольного количества чисел.

```
процедура min = функция (база ч)
   блок
   (ф64  число := ч  [ф64 : ] , % указатель вектора чисел
         m := число [0] ;
   для   k от 1 до длина число – 1 цикл
         если m > число [k] то m := число [k] все
         повторить ; m) % – –  – – %
```

Использование:

запф (*печ*, min (*a, в, c*, 1.0, 3.14))

1.17. Текстовые макросы

Макроаппарат языка Эль-76 имеет следующие преимущества перед более традиционными средствами макрогенерации:

– эффективность реализации;

– возможность использования текстовых макросов в разных контекстах – как выражения, оператора или переменной, а в некоторых случаях – и во всех трех этих функциях.

Наряду с процедурами тексты являются базовым средством структурирования программ, повышения их наглядности и надежности.

П р и м е р ы о п и с а н и й т е к с т о в.

```
текст  % тексты с параметрами:
     cand (a, в)  = % логическое "и" без "лишних" действий
       если a то в иначе ложь все,
     cor (a, в)  = % логическое "или" 'без "лишних" действий
       если a то истина иначе  в все

       % текст без параметров:
     конецблока =
       cor (длина буфблок = 0, буфблок [0] = 0);
```

Формальные параметры текста – идентификаторы. Правая часть описания текста – *выражение*, *переменная* или *оператор*, в зависимости от предполагаемого использования текста.

В описании текста *конецблока* дан пример подстановки текста *cor*. Фактическими параметрами могут быть любые предложения; при подстановке они неявно преобразуются в первичные. Подстановка текста реализуется как вставка объектного кода правой части, в который предварительно вставлены коды фактических параметров.

Иначе говоря, текстовый макрос языка Эль-76 – это *открытая процедура* (т.е. процедура, которая реализуется непосредственной подстановкой в место вызова).

Если в описании текста используется подстановка другого текста (например, подстановка текста *cor* в описании текста *конецблока*), то эта подстановка производится при трансляции описания использующего текста. Подстановка текста может входить и в фактический параметр под-

становки другого текста, например:

cor (*cand* (*a*, *b*), *c*).

В данном примере перед подстановкой текста cor предварительно выполняется подстановка текста *cand*.

Если правая часть описания текста обозначает переменную, то его подстановки, как и явные вхождения переменной, трактуются по-разному, в зависимости от места в программе: в выражении такая подстановка будет вычислять значение переменной, а в левой части присваивания – ее имя. Пример такого использования текста приведен ниже (пример 3).

Подстановка текста без параметров, в отличие от вызова процедуры, имеет вид: *конецблока* (скобки не требуются). При подстановке текста контролируется число фактических параметров, которое должно быть равно числу формальных. Не допускается возникновение рекурсии, т.е. в правую часть не может входить идентификатор определяемого типа. Обратим внимание еще на одну особенность: в правую часть могут входить глобальные идентификаторы, которые понимаются в том смысле, который они имеют в момент описания текста (аналогичным образом транслируются и описания процедур). Например:

```
конст eps = 1e-15;

текст равны (x, y) = % приближенное равенство для вещественных чисел
      модуль (x - y) < eps;
      начало
      конст eps = 1e-10;
        ... равны (a, в)  ... конец
```

При подстановке текста *равны* будет использовано первоначальное значение константы *eps*, равное 1*e*–15.

Применение текстов. С помощью текстов рекомендуется оформлять небольшие по объему фрагменты программы – модули (обычно объем текста не превышает 1–20 строк). Выделим наиболее распространенные случаи использования текстов.

1. Служебные фрагменты общего назначения, которые оформляются в виде текстов лишь для сокращения текста программы и улучшения ее наглядности:

```
текст натуральное (x) = cand (естьцел x, x > 0),
      меньше (s1, s2) = % лексикографический порядок строк
           если длина s1>= длина s2
           то s1< / s2
           иначе s1 <=/ s2
           все
```

2. Операции над сложными структурами данных. Здесь применение текстов позволяет скрыть особенности конкретного представления структуры данных (более подробно эти вопросы обсуждаются в гл. 5). В расширенном языке Эль-76 [52] для описания структур данных можно использовать

определяемые типы.

```
ф64 таблид := лок вект [начтаблид: макстаблид] ф64,
        % таблица идентификаторов
      уктаблид := 0; % текущий указатель таблид
поле % поля элемента таблид
    длинап = [63:8], % длина идентификатора
    индексп = [55:20], % индекс в таблице описаний
    следп = [35:20]; % связь в хеш-списке
текст % доступ к атрибутам элемента таблицы
    тдлина (т) = таблид [ т ] . длинап,
    тиндекс (т) = таблид [ т ] . индексп,
    тслед (т) = таблид [ т ] . следп,
    тидент (т) = таблид [ф8 т * 8+6:тдлина (т)];
      % указатель на символы идентификатора, которые
      % хранятся в элементе таблиц следом за значениями
      % полей
```

3. Использование одного и того же текста для обозначения как имени переменной, так и ее значения в зависимости от места в программе. Для процедур такой способ использования невозможен, так как процедура Эль-76 не может выдать имя в качестве результата.

```
% хранение нескольких логических переменных в одной
% простой переменной – основе
ф64 основа;
текст % описания логических переменных
   признак1 = основа. [63],
   признак2 = основа. [62]; % и т.п.
% использование:
признак1: = истина; ...
если признак2 то ...
```

4. Сокращение записи при вызове процедуры. Пусть, например, вызов процедуры *p* должен записываться в форме *p* (*x*, **имя** *y*), и программист желает обеспечить корректность формы вызова (т.е. не "потерять" спецификацию **имя**). Для этого следует воспользоваться вспомогательным описанием текста:

```
процедура pp = проц (a, в) ( ... );
   % описание процедуры p под другим именем
текст p (x, y) = % корректная форма вызова p
      pp (x, имя y);
% использование:
p (1, a); % правильный вызов
p (0);    % ошибка: неверное число параметров
```

При трансляции последнего вызова выдается сообщение о несовпадении числа фактических и формальных параметров текста *p*. При правильном вызове *p* спецификация **имя** подставляется в вызов процедуры *pp* автоматически.

Рассмотренные примеры демонстрируют гибкость аппарата текстовых макросов языка Эль-76. При использовании текстов, однако, следует учитывать возможность лавинообразного увеличения объектного кода из-за большого числа подстановок текстов. Такие особенности программы плохо поддаются формальному анализу и выявляются только при практическом использовании транслятора с Эль-76. Рекомендуется провести эксперимент по замене наиболее часто (с точки зрения программиста) используемых текстов на процедуры, не меняя формы обращения. Например, в первоначальном варианте текст может иметь вид:

```
текст новстрок = % перевод строки в файле протокола
      (запф (протокол, : l);
      если диалог то запф (экран, : l)
      все) % -- новстрок -- %
```

Вариант с заменой текста на процедуру:

```
процедура новстр = проц ( )
      (запф (протокол, : l);
      если диалог то запф (экран, : l)
      все);
текст новстрок = новстр ( )
```

Такая замена в данном случае не приведет к значительному ухудшению эффективности, так как действия модуля *новстрок* связаны с обменом. В практике программирования на Эль-76 известны случаи, когда такое преобразование программы приводило к сокращению объектного кода в полтора (!) раза.

1.18. Программа

Изучение различных видов структурирования программы в языке Эль-76 завершим изучением различных возможных форм самой программы. Здесь рассмотрены лишь базовые формы программы, включенные первоначально в ядро языка Эль-76 [52]. Расширения, связанные со средствами модульного программирования, описаны в п. 1.31. Формы внешних ссылок на программу и передаваемые ей файлы описаны в п. 1.29.3.

1.18.1. Программа-закрытое предложение. Простейшая форма программы – любое *закрытое предложение*, например:

```
начало % программы
   конст печ = читатрзадачи (виртвыв);
         % стандартный файл для выдачи на АЦПУ
   для k от 1 до 100 цикл
      запф (печ, 1/k, : l)
   повторить
конец
```

Здесь *запф* – операция форматного вывода, : *l* – перевод строки при выводе.

Как видно из примера, даже в этой простой программе печати первых 100 чисел, обратных натуральным, возникает вопрос о *стандартном файле вывода*. Доступ к этому файлу осуществляется конструкцией *читатрзадачи* (*виртвыв*). В примере стандартный файл вывода обозначен идентификатором *печ*.

Однако стандартного файла ввода в языке Эль-76 нет, так как программа может запускаться и с терминала, и в пакетном режиме. Поэтому более удобной формой программы в языке Эль-76 является такая, при которой программа, как процедура, имеет *параметры* (которые для большинства программ являются *файлами*).

1.18.2. Программа-определение процедуры. Программа, представленная в форме *определения процедуры*, имеет следующий вид:

```
печатьобратных = проц (ввод, вывод)
% —— печать чисел, обратных входным данным (до первого 0)
  (ф64 число;
  до финиш цикл
    читф (ввод, имя число); % ввод числа
    если число = 0 то финиш! % конец программы
    иначе запф (вывод, 1/число)
    все
  повторить ) % ——— печатьобратных ——— %
```

Программа, оформленная как определение процедуры, имеет свой *идентификатор* и *параметры*. Такая форма программы имеет некоторое сходство с программой на языке Паскаль. Однако, в отличие от языка Паскаль, в языке Эль-76 вопросы запуска программ, передачи им параметров, связи программ между собой и именования файлов не выходят за пределы языка, а являются его неотъемлемой частью.

1.18.3. Программа-пакет. Кроме программ-монолитов, требуется разработка и программ-пакетов, состоящих из описаний процедур, связанных общей информацией (константами, текстами и т.п.). Решение этой задачи в языке Эль-76 обеспечивается двумя способами. Если процедуры не имеют общих глобальных переменных, динамических констант, ситуаций и других общих описаний, приводящих к отведению памяти в области локальных данных программы, то пакет может быть описан с помощью конструкции **программа**, которая рассматривается ниже (в расширенном языке Эль-76 — конструкции "константный модуль" [52]). Если же у процедур имеются общие динамические объекты, следует воспользоваться *средствами модульного программирования* (п. 1.31).

Наличие двух различных языковых средств для организации пакетов объясняется историческими причинами. Средства модульного программирования появились в языке Эль-76 позднее как его расширение, более адекватно отвечающее потребностям разработки больших программ.

Указанное ограничение на общие данные пакета выполняется для достаточно большого класса пакетов процедур. Например, в виде программы-пакета оформлен стандартный пакет элементарных функций. В качестве примера рассмотрим схему описания пакета операций над векторами и матрицами.

```
программа
 % -- общие описания пакета -- %
    конст eps = 1e-14;
    текст модуль (x) = x. ( [63] : 0);
 % -- предварительные описания некоторых процедур пакета
    процедура ...
    скал1,             % скалярное умножение векторов
    сложматр1,         % сложение матриц
    умнматр1,          % умножение матриц
    печматр 1;         % печать матрицы
 % -- непосредственные описания процедур пакета -- %
    процедура скал1 = функция (x, y)
    до * неверныйоперанд, * границамассива
          если длина x < > длина y
          то недопустимыйаргумент!
          иначе
                ф128 c:= 0;
                для k до длина x-1 цикл
                c := c + x [ к ] * y [ к ]
                повторить; ф64окр c
                      все

    при неверныйоперанд:, границамассива: недопустимыйаргумент!

    всесит; % -- скал1 --- %
 % -- описания всех остальных процедур пакета -- %
 % -- описание константного справочника имен процедур: -- %
    спрконст (скал = скал 1, сложматр = сложматр1, ...
                умнматр = умнматр1, определитель = определитель1,
                обрматр = обрматр1, печматр = печматр1)
 конец % -- программы -- %
```

Конструкция **спрконст** называется *константным справочником* и служит для объединения элементов пакета (в данном случае – процедур) в одну структуру под заданными именами. Элементы справочника задаются списком пар: *имя* = *значение*. Имя в левой части задает тот идентификатор, под которым процедура будет доступна как компонента пакета. В правой части указывается значение элемента (метка процедуры).

Использование конструкции **программа**, в сочетании с константным справочником, указывает системе, что данную программу следует рассматривать как *пакет*, так как константный справочник является *результатом открытия* программы (подготовки ее к исполнению). Открытие программы выполняется операцией *прогр* (*k*), где *k* – ссылка на файл объектного кода программы. Результатом этой операции для программы-закрытого предложения и программы-процедуры является процедурная метка для ее запуска, для программы-пакета – константный справочник. Для сравнения, обычная программа может быть записана следующим образом:

```
программа проц ( ) ( p ) конец,
```

где *р* – тело программы. В данном случае результатом открытия является *метка процедуры.* Наиболее простой способ использования программы-пакета состоит в следующем. Пусть объектный код пакета записан в файл с внешним именем *//матрицы* (п. 1.23.2). Подключение процедур пакета осуществляется с помощью операции *прогр.* Имя процедуры из константного справочника задается конструкцией "внешнее имя" (п. 1.23.2):

```
начало
% подключение процедур пакета:
конст
    сложматр = прогр ( // матрицы // сложматр),
    печматр = прогр ( // матрицы // печматр);
    конст м1 = лок вект [10, 10] ф64,
          м2 = лок вект [10, 10] ф64,
          м3 = лок вект [10, 10] ф64;
  . . .
    сложматр (м1, м2, м3);
    печматр (м3)
конец
```

В этой программе можно использовать идентификаторы процедур пакета, заданные с помощью константного справочника в пакете *// матрицы.*

1.19. Форматный обмен

Средства форматного обмена (в более привычной терминологии – ввода-вывода) составляют последнюю группу конструкций, которую можно отнести к ядру языка Эль-76. Стандартные процедуры форматного обмена Эль-76 работают с *текстовыми файлами.* Текстовый файл связан с одним из типов *внешних устройств*, пригодных для обработки текстовой информации. Особенности организации текстовых файлов рассмотрены в п. 1.27. Здесь отметим лишь основные свойства текстовых файлов, существенные для форматного обмена. Текстовый файл логически делится на *строки*, каждая строка состоит из *символов.* Поскольку форматный обмен выполняется последовательно, процедура форматного обмена хранит и модифицирует *текущую позицию* по файлу – ссылку на текущую строку и ее первый необработанный байт. Для представления информации о текущей позиции используется *позиционная переменная* – один из видов *оперативных объектов связи с файлом* (п. 1.20.3). В наиболее распространенных случаях позиционная переменная файла для выполнения операций форматного обмена создается следующим образом.

Если программа имеет вид определения процедуры:

программа = **проц** (*вв, выв*) (. . .),

где *вв* и *выв* – ссылки на входной и выходной текстовые файлы (п. 1.27), то позиционные переменные этих файлов создаются операцией *позп*:

```
конст ввод = позп (вв),
      вывод = позп (выв)
```

Стандартная операция *открел* служит для уничтожения (открепления) позиционных переменных до завершения программы, например:

открел (*ввод*); *открел* (*вывод*)

При завершении всей программы эта операция выполняется неявно (для стандартного файла вывода – при завершении задачи).

2. Позиционная переменная стандартного файла виртуального вывода задачи является результатом операции *читатрзадачи*:

конст *печ* = *читатрзадачи* (*виртвыв*)

Операции форматного обмена в языке Эль-76 представлены в виде стандартных процедур:

запф (*поз*, $e_1, \ldots, e_n$)
запфм (*стр*, $e_1, \ldots, e_n$)
читф (*поз*, $v_1, \ldots, v_n$)
читфм (*стр*, $v_1, \ldots, v_n$)

Операции *запф* и *читф* осуществляют запись в текстовый файл и чтение из текстового файла. Их первым параметром должна быть позиционная переменная на этот текстовый файл. Параметры e_i – *элементы вывода* (*выражения* с указанием или без указания формата), v_i – *элементы ввода* (*переменные* с указанием или без указания формата). В зависимости от наличия формата, традиционно различают *обмен, управляемый данными* (по стандартному формату), и *обмен, управляемый форматом.* В одной операции обмена могут использоваться обе формы для разных элементов.

Элемент ввода или вывода может также иметь вид : *а*, где *а* – *позиционирование* (установка текущей позиции). Например, :*е* – перевод строки.

Операции *запфм* и *читфм* предназначены для обмена с байтовой строкой, рассматриваемой как текстовый файл из одной строки. Первым параметром этих процедур должна быть *стр* либо конструкция **имя** *с* (где *с* – переменная, значение которой – указатель байтовой строки). В последнем случае после завершения форматного обмена переменная *с* модифицируется указателем на необработанную часть строки. Операции *запфм* и *читфм* полезны для упрощения реализации лексического анализа текста. Например, если символьное представление вещественного числа записано в буфер *буф* (с помощью операций п. 1.6.6) и имеет длину *дл*, то операция

читфм (*буф* [: дл], **имя** *x*)

переводит число во внутреннее представление и присваивает его значение переменной *x*.

Ситуации при форматном обмене. Исчерпание входного файла при чтении приводит к возникновению стандартной ситуации *ситлкф* (логический конец файла). При превышении физических границ выходного файла при записи возникает стандартная ситуация *ситфкф* (физический конец файла). Эти ситуации можно обработать с помощью структурных предложений (п. 1.15.3). В качестве примера рассмотрим процедуру переписи содержимого входного файла в выходной (в предположении, что

входной файл состоит из чисел)

```
процедура перепись = проц (откуда, куда)
    начало
       конст ввод = позп (откуда),
             вывод = позп (куда),
             печ = читатрзадачи (виртвыв);
       ф64 x;
       до * ситлкф, * ситфкф
       цикл
           читф (ввод, имя x);
           запф (вывод, x, : l). % каждое число – на новую строку
       повторить
       при ситфкф:
           запф (печ, стр8 "переполнение выходного файла")
       всесит;
       откреп (ввод); откреп (вывод)
    конец % --- перепись --- %
```

1.19.1. **Обмен, управляемый данными.** При этом способе обмена обеспечивается автоматический перевод строки при вводе и выводе и разделение выводимых подряд чисел пробелами. Однако он менее удобен тем, что для чисел выводится всегда максимально возможное число позиций.

В ы в о д. Примеры:

1. *запф* (*печ*, **цел32** 1, –2, **вещ32** 3.0, 1*e*–10)

В выходной файл будет выведена следующая информация (символ "␣" обозначает пробел; разделение на строки условное, так как все числа выводятся в одну строку):

␣␣␣␣␣␣␣␣␣+1
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣–2
+ 3.0000000*e* + 1.0000000000000000*e* – 10

Для целых 32 выводятся знак и 10 цифр, для целых 64 – знак и 19 цифр (если значащих цифр меньше, число дополняется слева пробелами). Вещественные выводятся в плавающей форме: для вещественных 32 – одна цифра целой части, 7 – дробной, 2 – порядка; **вещ64** – 1, 16 и 2, **вещ128** – 1, 33 и 5. Числа автоматически разделяются пробелами.

2. *запф* (*печ*, "a", *мконст* **ф32** (**цел32** 1, **цел32** 2, **цел32** 3),
стр8 "строка", **стр1** "1101111")

Результат вывода:

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣+193␣␣␣␣␣␣␣␣␣
+1␣␣␣␣␣␣␣␣␣␣+2␣␣␣␣␣␣␣␣␣␣+3
строка␣1101111

Набор выводится как целое 64 (набор "a" – как +193, так как код символа "a" равен 193); вектор – поэлементно; строка – в виде последовательности символов, изображающих ее элементы (в зависимости

от формата). Многомерный массив выводится по строкам. Значения **пусто32** и **пусто64** выводится в виде последовательности звездочек. Значения других типов выводить нельзя (будет выдано сообщение об ошибочном вызове процедуры *запф*).

В в о д. Пусть выполнены следующие описания:

ф32 *i*; **ф64** *x*;
конст *в* = **лок вект** [10] **ф64**,
с = **лок вект** [10] **ф8**
1. *читф* (*ввод*, **имя** *i*, **имя** *x*, **имя** *с* [:3]@, *в* [2:2], *с* [3:])

Содержимое входного файла:

␣␣␣–256␣␣3.14*e*␣–1␣0␣–2.718␣11␣строка

Результат ввода: *i* ⇔ – **цел32** 256; *x* ⇔ 0.314; *с* [:3]@ ⇔ 4"000000"; *в* [2] ⇔ –2.718; *в* [3] ⇔ 11; *с* [3:]@ = "␣строка"

Перед вводимым числом допускаются пробелы и концы строк. Число набирается до ближайшего пробела или другого ограничителя. Вектор вводится поэлементно; строка – как последовательность элементов, длина которой равна длине строки (они должны изображать биты, цифры или символы). В случае неверных исходных данных (неверного синтаксиса числа или слишком большой его величины для формата переменной) простой переменной присваивается пустой объект, а упакованная переменная заполняется нулевыми битами.

1.19.2. Обмен, управляемый форматом. Формат вводимого (выводимого) значения указывается для каждого элемента ввода (вывода) отдельно; форма записи близка языку Паскаль. Специального типа данных "формат", как в Алголе-68, в языке Эль-76 нет.

Формат состоит из необязательной *вставки* и *трафарета*. Вставка задает текст, который вводится (выводится) перед значением. Например:

запф (*печ*, *x*: "x=" *g* (5))

Вывод: *x* = ␣␣␣ + 1.

Т р а ф а р е т ы. Имеются специальные трафареты для ввода и вывода значений различных типов: *трафарет числа, трафарет целого, трафарет литерного, трафарет двоичного, трафарет тегированного.* Рассмотрим более подробно первый из них как наиболее употребительный, а остальные – на примерах (более подробная информация приведена в [52]).

Трафарет числа имеет вид:

g (*m*) или *g* (*m*, *n*) или *g* (*m*, *n*, *p*)

Здесь $|m|$ – общее число позиций (символов), *n* – число позиций дробной части, *p* – число позиций порядка. Если $m < 0$, то знак учитывается только для отрицательных чисел; при $m = 0$ выводится минимальное число позиций. При *вводе* по трафарету числа значения *m*, *n* и *p* игнорируются, и число вводится как при обмене, управляемым данными.

П р и м е р ы.

1. *запф* (*печ*, 100; "x=" *g* (–3), 20.75: "y=" *g* (6,2), 1.36*e*–15: " z=" *g* (9,2,3))

вывод:
x=100␣␣␣ y=20.75␣␣␣ z= ␣␣␣ 1.36e–15

2. Трафареты целого и вещественного:

запф (*печ*, 1234567: "телефон:" 6*zd*, 15.3: " скорость:" –*zd.d*)

вывод:
телефон: 1234567␣␣␣ скорость: ␣␣␣ 15.3

Трафареты целого и вещественного очень близки по своим возможностям соответствующим форматам языка Алгол-68. В отличие от трафарета числа при вводе по этим трафаретам все размеры их компонент должны соблюдаться.

3. Трафареты логического, литерного, двоичного и тегированного:

запф (*печ*, **истина**: *в*, " набор ": 7*а*, 4"*FF*": 4*r* 2*d*" ", **пусто64**: 4*t*)

вывод:

и ␣␣␣ набор␣*FF*␣20␣0000000000000000

По трафарету логического выводится буква "и" или "л"; по трафарету литерного – последовательность символов набора или строки; по трафарету двоичного – битовое представление в двоичном (1*r*), восьмеричном (3*r*) или в шестнадцатеричном (4*r*) виде; по трафарету тегированного – тег в шестнадцатеричном виде и значение в битовом представлении (по трафарету тегированного можно выводить любое значение).

1.19.3. Позиционирование. Позиционирование в процедурах форматного обмена предназначено для управления текущей позицией и для вывода заданной последовательности символов. Элемент-позиционирование задает последовательность *кодов размещения* (действий над текущей позицией), каждый из которых может иметь повторитель. Различаются следующие коды размещения:

x – сдвиг вправо на один символ (при выводе – вывод пробела);

y – сдвиг влево на один символ;

k – сдвиг на байт с указанным номером в текущей строке (например, 1*k* – сдвиг на первый символ);

l – перевод строки.

Пример:

запф (*печ*,: "автокод" 7y "Эль-76" *xl*)

Вывод: Эль-76␣

Форматный обмен в практике программирования на Эль-76 используется не только для ввода-вывода в традиционном смысле, но и для организации диалогового режима работы программы (п. 1.26). Форматный вывод удобен также для автоматической генерации фрагментов программы – например, постоянных таблиц и битовых шкал, которые наиболее удобно вычислять программным путем. Содержимое таблицы, полученной во вспомогательной программе, может быть с помощью процедур форматного обмена выведено в заданный текстовой файл и затем средствами текстового редактора (п. 1.33.7) символьное представление таблицы вставляется в текстовой файл основной программы. Например, если таблица *табл* представлена в виде вектора формата **ф64**, а // *табл* – внеш-

нее имя (п. 1.23.2) текстового файла для вывода таблицы, то с помощью форматного вывода в файле // *табл* генерируется представление таблицы в виде текста изображения массива констант, каждый элемент которого представлен в шестнадцатеричном виде:

```
(ф64 n: = позп (//табл),
      длтаб: = длина табл;
запф (n,: "конст табл = _мконст_ф64 ("l);
для i до длтаб-1 цикл
      запф (n, "4"""": 2a, табл [i] : 4r16d,
            если i = длтаб–1% последний элемент
            то """"); "
            иначе """", "
            все: 3a,: l)
повторить;
откреп (n)
)
```

1.20. Основные понятия системы простых файлов

1.20.1. Структура и назначение системы файлов. Средства управления внешними объектами (объектами, расположенными на внешних устройствах) объединены в МВК "Эльбрус" в *систему файлов.* Язык Эль-76 – базовый язык программирования МВК "Эльбрус" – имеет стандартное расширение, с помощью которого программисту доступны все возможности системы файлов. Выделены три уровня взаимодействия с внешними объектами:

– *физический обмен* – действия над конкретными внешними устройствами. Эта группа операций используется в основном для реализации следующего уровня системы файлов и в данной работе не описывается;

– *система простых файлов* (СПФ) – система понятий, объектов и операций, объединяющая базовый системный уровень взаимодействия с внешними объектами, а также средства для реализации обмена в языках программирования;

– *система структурированных файлов* (ССФ) – совокупность операций, обеспечивающая работу с внешними объектами на уровне их логической структуры, методов доступа и запросов к информации (этот уровень приблизительно соответствует СУБД). ССФ реализованы с использованием СПФ.

Поскольку система структурированных файлов появилась совсем недавно, в течение нескольких лет на МВК "Эльбрус" эксплуатировалась главным образом система простых файлов. Именно с ней связан приобретенный нами опыт программирования операций над внешними объектами, поэтому в этой книге СПФ уделено основное внимание.

Следует признать, что этот опыт был как положительным, так и отрицательным, что объясняется разнообразными и противоречивыми требованиями предъявляемыми к СПФ различными классами задач и категорий программистов. С одной стороны, СПФ оказалось достаточно гибкой и мощной для реализации систем обмена в языках программирова-

ния (Фортран, Кобол, ПЛ/1, Клу, Алгол-68). С другой стороны, при разработке системы программирования ощущалась потребность в средствах работы с записями, которые появились значительно позже, только в системе структурированных файлов. Прежде всего это относится к поддержке обмена логическими записями, характерного для большинства языков программирования. Ее отсутствие в СПФ привело к тому, что эти средства моделировались, по существу, для каждого языка отдельно.

При применении СПФ прикладными программистами на языке Эль-76 многие элементы СПФ (средства управления атрибутами, различные формы ссылок на внешние объекты и операции обмена), хотя они и предоставляют программисту дополнительные возможности, по сравнению с более традиционными системами файлов оказались менее привычными, чем средства обмена языков Фортран, Алгамс, ПЛ/1, Паскаль.

В СПФ достигнута значительно большая степень унификации представления внешних объектов и основных операций над ними. В системе ОС ЕС, в отличие от СПФ, для наборов данных (файлов) имеются несколько различных, не связанных между собой методов доступа; в системе UNIX отсутствуют средства управления атрибутами. СПФ обладает бо́льшим динамизмом и гибкостью. Она распространяет общий динамический подход, лежащий в основе архитектуры системы, на внешние объекты. В СПФ внешние объекты создаются и ликвидируются динамически (исполняемыми конструкциями Эль-76), по мере необходимости. Тот же принцип применен и при распределении памяти для внешних объектов. Отсутствует характерная для традиционных файловых систем жесткая связь внешнего объекта с идентификатором (именем *DD*-предложения в языке управления заданиями, именем элемента в текущем справочнике пользователя и т.п.). Это позволяет снять статические ограничения на число создаваемых внешних объектов. Более гибкими, чем традиционные, являются и архивные механизмы СПФ, обеспечивающие создание произвольного сетевого внешнего контекста и защиту от несанкционированного доступа к внешним объектам. Подтверждением гибкости СПФ является тот факт, что она используется как для нужд ОС, так и в обычных системных и прикладных задачах.

Включенное в настоящую работу руководство по системе простых файлов МВК "Эльбрус" (пп. 1.20–1.29) не является полным. В нем рассмотрены лишь те конструкции, которые наиболее часто использовались группой разработчиков под руководством автора.

1.20.2. Внешние объекты. Внешние объекты в СПФ могут быть следующих типов: *файлы, контейнеры* и *справочники*. Они соответствуют различным способам использования внешней памяти.

Более традиционным понятием является *файл*. Файл – это совокупность данных, связанных с одним из допустимых типов внешних устройств: *барабан, диск, лента, алфавитно-цифровой дисплей, АЦПУ, устройства ввода-вывода перфокарт и перфолент, пишущая машинка* и т.д. Файл состоит из *заголовка* и *памяти*. В заголовке хранятся *атрибуты* файла (например, *тип внешнего устройства*) и ссылка на его память. В памяти файла хранятся элементы данных.

Контейнер – это область внешней памяти для хранения файлов и справочников. Контейнер, как и файл, состоит из *заголовка* и *памяти* и имеет

ряд *атрибутов*, в частности *имя*. Контейнеры организуют на *барабанах* (системный контейнер *барабан* объединяет всю доступную память на барабанах), *дисках* и *лентах*. Количество томов (носителей на дисках или лентах) одного контейнера не ограничено и при необходимости автоматически увеличивается системой. Любой файл на барабане, диске или ленте всегда создается в некотором контейнере.

Справочник предназначен для организации *архива*. Наиболее распространенный вид справочника – это каталог именованных ссылок на внешние объекты. С помощью справочника эти объекты получают символьные обозначения. Количество элементов в справочнике не ограничено. Как сам справочник, так и его элементы имеют свои *атрибуты*. В отличие от традиционных библиотечных наборов данных ЕС ЭВМ сами именуемые внешние объекты размещаются отдельно от справочника и связываются с ним ссылками. Близкое понятие справочника (directory) введено в ОС UNIX [11].

1.20.3. Оперативные объекты связи. При организации объема используются различные связи внешнего объекта с исполняемой программой. Для этого создаются специальные объекты в оперативной памяти – *оперативные объекты связи*. Имеются следующие типы этих объектов, каждый из которых соответствует определенному способу обмена.

1. *Заголовок открытого файла* предназначен для организации *непосредственного обмена* с файлом – наиболее простого способа обмена, при котором содержимое части файла переписывается в некоторый вектор (либо содержимое вектора – в файл) без какой-либо буферизации (пп. 1.24.2, 1.25.2, 1.26.2). Обмен с помощью заголовка открытого файла выполняется *синхронно* (программа ждет окончания обмена).

2. *Позиционная переменная* служит для *однобуферного обмена* с файлом – наиболее удобного способа обмена, при котором в каждый момент некоторая часть (*блок*) файла доступна через указатель *буфера*, содержащего элементы блока (пп. 1.24.3, 1.25.3, 1.26.3).

3. *Блок ввода-вывода* используется при *параллельном непосредственном обмене*. Такой обмен выполняется параллельно с запустившей его программой пользователя, которая может при необходимости в некоторой точке ожидать окончания обмена с помощью специального семафора. Блок ввода-вывода неявно формируется также при создании заголовка открытого файла и позиционной переменной.

4. *Документация контейнера* используется при создании и обработке файлов в данном контейнере.

5. *Таблица буферов* применяется при *многобуферном обмене*, при котором одновременно доступно произвольное число блоков файла. Этот способ обмена и объект связи в данной работе не рассматриваются (см. [52]).

6. *Подвижный указатель внешнего объекта* играет роль ссылочной переменной для настройки на различные внешние объекты. Он не связан с каким-либо способом обмена, а используется для ссылки на внешние объекты (например, при их открытии или модификации атрибутов) или для управления архивом (пп. 1.23.1, 1.29.3).

7. *Массив процедур реакций на ошибки обмена* представляет пользователю возможность нестандартным образом реагировать на возможные

ошибки обмена. После завершения реакции в некоторых случаях обмен может быть продолжен. Этот объект связи и способ обработки ошибок обмена также не рассматриваются в данной работе, так как чаще всего применяется обработка ошибок обмена с помощью структурных предложений (п. 1.15).

1.20.4. Период существования и ликвидация внешних объектов. Любой внешний объект может быть локальным или глобальным. Имеется возможность управления локализацией. Внешний объект становится глобальным, если ссылка на него записана в архив (в какой-либо справочник), и существует до тех пор, пока на него есть хотя бы одна ссылка из какой-либо задачи или из архива. Для контроля связей внешнего объекта применяется метод счетчиков ссылок. При создании новой ссылки на объект его счетчик ссылок увеличивается на 1, при ликвидации ссылки – уменьшается на 1. При обнулении счетчика ссылок внешний объект уничтожается. Ликвидация ссылки на объект выполняется операцией *ликвидэлспр* (*с*), где *с* – ссылка на элемент справочника. Операция *ликвидвнеш* (*с*) уничтожает внешний объект без учета ссылок. Локальный внешний объект существует до завершения блока, в котором он был создан

1.21. Файл: создание, открытие, закрытие, атрибуты

В традиционных файловых системах (ЕС ЭВМ, СМ ЭВМ) файл (набор данных) создается конструкцией языка управления заданиями и связывается с определенным идентификатором, по которому он впоследствии доступен из программы на обычном языке программирования. В системе "Эльбрус" принят более динамичный подход. Файл создается в программе на Эль-76 с помощью *генератора.* Статических ограничений на число создаваемых файлов нет. Созданный файл сам по себе не связан с каким-либо символьным именем. Ссылка на файл может быть, например, значением локальной переменной. Для сохранения ссылки на файл после завершения программы с помощью операций над архивом можно создать для него *внешнее имя,* включив его в некоторый справочник в том же контейнере (п. 1.23.2).

1.21.1. Генератор файла имеет вид:

генфайл (*конт, атрибуты*)

или:

генфайл (*ув, конт, атрибуты*)

(первая форма используется чаще). Здесь: *конт* – ссылка на контейнер, в котором создается файл (п. 1.22), или стандартный идентификатор, задающий тип внешнего устройства; *атрибуты* – значения атрибутов файла (при отсутствии выбираются по умолчанию). Результат генератора – ссылка (указатель) на файл.

Вторая форма генератора настраивает на создаваемый файл заданный подвижный указатель *ув;* результатом в этом случае является значение *ув.*

Примеры.

1. Создание файла на барабане. Все барабанные файлы создаются в системном контейнере *барабан*:

ф1: = *генфайл* (*барабан*)

Ссылка на файл здесь присваивается переменной *ф1*, которая в дальнейшем используется для обозначения файла. Барабан используется главным образом для создания *временных* файлов.

2. Создание файла на диске выполняется операцией *генфайл* (*к, атрибуты*), где *к* – ссылка на контейнер, в котором создается файл. В примере показано создание файла в контейнере с внешним именем *ксн//лгу* (п. 1.23.2). При создании указано значение атрибута *типфайла* (3 – произвольный текст):

ф2: = *генфайл* (*ксн//лгу, типфайла*: 3)

3. Создание файла на ленте, в контейнере с именем "сброс". Перед созданием файла контейнер открывается, и лента устанавливается на начало (подробнее см. п. 1.25). Созданный файл получает имя "текст":

л: = *контейнер* (*ксн//сброс*);
ф3: = *генфайл* (*л, имяфайла*: "текст")

Файл на ленте всегда создается в той позиции, куда установлена лента. Созданный на ленте файл является всегда *последним*, т.е. создание файла в некоторой позиции ленты приводит к уничтожению всех последующих файлов.

Создание файлов на других носителях. В некоторых случаях, ввиду особенностей внешних устройств, файл не может быть создан, а может быть только открыт, так как его физический прообраз уже установлен на устройство. Это относится к файлам для чтения с перфокарт и с перфоленты (*чпк, чпл*). Кроме файлов на барабане, диске и ленте возможно создание файлов для вывода на АЦПУ (*ацпу*) на перфокарты (*зпк*) и перфоленты (*зпл*). Однако все эти возможности используются редко, так как для вывода на АЦПУ чаще всего используется *стандартный файл виртуального вывода* задачи, а для переноса информации с установки на установку и длительного хранения информации используются в основном ленты.

Особых пояснений требуют файлы, связанные с *алфавитно-цифровым дисплеем* (*ацд*), предназначенные для организации *диалога* (п. 1.26). При входе пользователя в диалог с системой запускается новая (диалоговая) задача, в рамках которой для ведения диалога система создает терминальный файл. Создание файлов на дисплее с помощью конструкции *генфайл* невозможно.

1.21.2. Открытие и закрытие файла. Для выполнения операций обмена с любым типом внешних устройств файл прежде всего должен быть открыт. *Открытие файла* – это копирование его заголовка в оперативную память и подготовка к обмену с файлом (*создание оперативного объекта связи*). Если файл используется в нескольких задачах, то заголовок файла хранится в памяти в единственном экземпляре, но для каждой задачи задается свой объект связи. Синхронизация обращений к файлу возлагается на пользователя и поддерживается системой с помощью семафора, хранящегося в заголовке.

1. Открытие файла для *синхронного непосредственного обмена* (пп. 1.24.2, 1.25.2, 1.26.2):

з: = *файл* (*ф*)

Переменная *з* содержит указатель на *заголовок открытого файла*.

2. Открытие файла для *параллельного непосредственного обмена* (п. 1.25.4):

б := *бвв* (*ф*)

Переменная *б* содержит указатель на *блок ввода-вывода*. Для организации синхронного обмена блок ввода-вывода создается неявно и связывается с заголовком открытого файла.

3. Открытие файла для *однобуферного обмена* (пп. 1.24.3, 1.25.3, 1.26.3):

п := *позп* (*ф*)

или

п := *позп* (*з*)

Переменная *п* содержит указатель на *позиционную переменную*. Если в операцию подан закрытый файл *ф*, выполняется его неявное открытие.

Закрытие файла (*открепление*) – это ликвидация оперативного объекта связи с файлом. При закрытии выполняются все незавершенные действия по обмену с файлом, – например, в файл записывается последний обновленный блок. Для любого вида оперативного объекта связи *х* закрытие (более точно – ликвидация оперативного объекта связи) выполняется операцией:

откреп (*х*)

Для своевременного завершения обмена рекомендуется операцию *откреп* выполнить явно. Неявно она выполняется по завершении блока, в котором создан объект связи.

1.21.3. Атрибуты файла. В СПФ введена гибкая система атрибутов файла (порядка нескольких десятков) для чтения и модификации его характеристик, управления обменом и распределением памяти. Каждый атрибут имеет стандартное *имя*; значение атрибута может задаваться при генерации файла или выбираться по умолчанию. Для *вычисления значений атрибутов* служит операция

читатр (*ф*, *имя*)

где *ф* – ссылка на файл, *имя* – имя атрибута. *Модификация атрибутов* выполняется операцией:

запатр (*ф*, *имя1*: *значение1*, . . . , *имяк*: *значениек*)

Для всех типов устройств определены следующие атрибуты файлов:

1. *типву* – тип внешнего устройства: 1 – барабан, 2 – диск, 3 – лента, 4 – ввод с перфокарт, 5 – вывод на перфокарты, 6 – ввод с перфоленты, 7 – вывод на перфоленту, 8 – алфавитно-цифровой дисплей, 9 – пишущая машинка, 10 – АЦПУ. Атрибут *типву* фиксируется при создании файла и не подлежит модификации.

2. *типфайла* – тип информации, хранящейся в файле. Этот атрибут является аналогом тега для внешних объектов и используется в ОС для контроля. Возможны следующие его значения:

0 – *данные*: произвольная двоичная информация (например, промежуточные результаты вычислений в двоичном виде); файл с этим типом не подлежит редактированию и другим действиям, применимым к текстовым файлам;

1 – *псевдофайл*: файл, моделируемый на устройстве другого типа; например, этот тип имеет файл виртуального вывода, размещаемый на барабане;

2 – *файл объектного кода* (ФОК): выходной файл компилятора, результат трансляции программы с какого-либо языка;

3 – *произвольный текст*: текстовой файл стандартной структуры (п. 1.27), содержащий любой текст – программу, документацию и т.п.; это значение атрибута *типфайла* устанавливается по умолчанию;

4, 5, 6 – *текст программы на языке Эль-76* (4), *Алгол* (5), *Фортран* (6). Значение атрибута *типфайла* контролируется компиляторами: например, компилятор Эль-76 воспринимает текст со значением атрибута *типфайла*, равным 3 (произвольный текст) или 4 (текст на Эль-76).

Атрибут *типфайла* можно модифицировать, изменяя тем самым трактовку записанной в файл информации. Например, операция

запатр (*ф*, *типфайла*: 0)

для файла объектного кода *ф* предохраняет его от запуска, а для текста – от редактирования.

3. *максдлинблока* – максимальная длина блока при буферизованном обмене (пп. 1.24.3, 1.25.3, 1.26.3); для барабана и диска – в словах, для других типов устройств – в байтах. Атрибут можно модифицировать, если файл закрыт, например:

```
ф := генфайл (барабан, максдлинблока: 512);
фпоз := позп (ф);
% обмен с файлом ф (длина блока – 512 слов)
открeп (фпоз);
запатр (ф, максдлинблока: 256);
фпоз := позп (ф);
% обмен с файлом ф (длина блока – 256 слов)
открeп (фпоз)
```

4. *максдлинфайла* – условная верхняя граница файла, введенная для контроля его размера. Измеряется в единицах распределения внешней памяти: для дисков и барабанов – в *листах* (п. 1.24), по умолчанию – 10; для остальных типов устройств – в байтах (по умолчанию – 999999). При превышении значения этого атрибута возникает ситуация *ситфкф*, но его значение может быть увеличено, например:

```
до конецобмена цикл
   до * ситфкф
        (цикл
        % – – запись в файл ф – – %
        повторить;
        конецобмена!)
   при ситфкф:
        запатр (ф, максдлинфайла:
                    читатр (ф, максдлинфайла) + 10)
   всесит
повторить
```

Атрибуты, присущие файлам на устройствах других типов, описаны ниже.

1.22. Контейнер: основные операции и атрибуты

Контейнеры для размещения файлов и справочников создаются на барабанах, дисках и лентах. Свободная память на барабане объединена в системный контейнер *барабан*. Контейнер на диске создается *генератором контейнера*:

к := *генконтейнер* (*мдконт*, *имяконт*: *имя*)

Здесь *мдконт* – стандартный идентификатор, обозначающий создание контейнера на диске; *имяконт* – атрибут, задающий имя контейнера, *имя* – имя контейнера (строка или набор, например "лгу"). Контейнер на ленте создается аналогичным образом:

к := *генконтейнер* (*млконт*, *имяконт*: "лгу")

В обоих случаях выполняется разметка носителя, контейнеру присваивается указанное имя, которое записывается в его метку.

Созданный контейнер на диске доступен по *внешнему имени* (в примере – *ксн*//*лгу*); *ксн* – *контекст съемных носителей* (п. 1.23.2). Внешнее имя – основная форма ссылки (на внешний объект) на контейнер. Вместе с контейнером на диске создается его *базовый справочник* для именования создаваемых в нем объектов (п. 1.23.2).

Структура контейнера на ленте более простая: он состоит из начальной и конечной меток, между которыми находятся созданные в нем файлы. Особенности работы с контейнерами на лентах описаны в п. 1.25.

Открытие контейнера выполняется операцией:

контейнер (*к*),

где *к* – ссылка на контейнер. Результат – *документация контейнера*. Открытие контейнера имеет существенное значение для магнитных лент, так как при открытии можно указывать позиционирование ленты на заданный файл. При создании файлов в контейнере на диске можно задавать ссылку на закрытый контейнер (например, внешнее имя). Если требуется, система выполняет неявное открытие контейнера. Контейнер *барабан* считается всегда открытым.

Закрытие контейнера выполняется операцией *откреп* (*ок*), где *ок* – ссылка на открытый контейнер. Операция закрытия особенно существенна для контейнеров на лентах, так как при открытом контейнере на ленте с контейнером может работать только одна задача, открывшая контейнер. Закрытие контейнера выполняется неявно при входе из блока, в котором было выполнено его открытие.

Атрибуты контейнера. Контейнер имеет два основных атрибута: *типву* – тип внешнего устройства (1 – барабан, 2 – диск, 3 – лента) и *имяконт* – имя контейнера. Чтение атрибута *имяконт* имеет свои особенности: в операции *читатр* требуется задавать указатель байтовой строки, в котором система размещает символы имени и выдает указатель подмассива, содержащего эти символы. Например:

конст *с* = **лок вект** [20] **ф8**,
    имя = *читатр* (*ксн*//*лгу*, *с*, *имяконт*);
запр (*печ*, *имя*) % в файл *печ* выведено: *лгу*

Контейнер имеет также ряд атрибутов для управления его томами (дисками, лентами). Однако чаще всего контейнер содержит только один том. При его исчерпании система выдает запрос оператору о необходимости установки носителя для нового тома данного контейнера.

1.23. Справочники. Архив

Справочник – это каталог ссылок на внешние объекты, имеющих свои *имена* или *номера*. Справочники могут создаваться только на диске или на барабане. Элементы справочников ссылаются на файлы или другие справочники; иерархия или сеть справочников образует *архив*.

1.23.1. Создание и элементы справочника. Подвижные указатели. Создание справочника, как и других внешних объектов, выполняется с помощью *генератора*: *генспр* (*основа*) или: *генспр* (*ув*, *основа*), где *ув* – подвижный указатель для настройки на созданный справочник; *основа* – ссылка на контейнер, где создается справочник. Например:

с := *генспр* (*барабан*) % справочник на барабане

или

с := *генспр* (*ксн*//*лгу*) % справочник на конкретном диске

При создании справочника не фиксируются какие-либо его атрибуты, в частности, количество элементов. При необходимости справочник автоматически расширяется.

Создание элемента справочника. Если *с* – справочник, *ф1* – ссылка на файл, то операция:

создэлспр (*с*, "ф1", *ф1*)

создает новый элемент справочника с именем "ф1" и ссылкой на файл *ф1*. Элементы справочника хранятся *в алфавитном порядке* их имен.

Обращение к элементу справочника. Подвижный указатель. Для настройки на элементы справочника используется *подвижный указатель внешнего объекта*. Он создается операцией *генув* ( ). Подвижный указатель должен быть задан первым параметром в операции:

элспр (*ув*, *с*, *имя*)

Здесь *ув* – подвижный указатель, например конструкция *генув* ( ); *с* – справочник; *имя* – имя элемента справочника (строка или набор). Операция выполняет поиск заданного элемента и настройку на него подвижного указателя *ув*. Результат – настроенный указатель *ув*.

Если элемент с данным именем отсутствует, возникает стандартная ситуация *ситнетварх* (нет в архиве).

После выполнения операции

х := *элспр* (*генув* ( ), *с*, "ф1")

переменная *х* может использоваться для ссылки на файл *ф1* во всех операциях над ним, например:

поз := *позп* (*х*)

Таким образом, ссылка через подвижный указатель является второй возможной формой ссылки на внешний объект.

Изменение значения элемента справочника осуществляется операцией:

запспр (*эл*, *знач*)

где *эл* – указатель на элемент справочника, *знач* – новое значение элемента. Например, выполнение операции

запспр (*элспр* (*генув* ( ), *с*, "ф1"), *ф2*)

приводит к тому, что ссылка на файл *ф1* из элемента справочника *с* заменяется ссылкой на файл *ф2*.

1.23.2. Внешнее имя. План поиска. Понятие справочника дает еще одну удобную форму ссылки на файл – *внешнее имя*. В рассматриваемом примере вместо операции *элспр* для ссылки на файл *ф1* можно использовать его внешнее имя:

с//*ф1*

Отличие этой конструкции от операции *элспр* в том, что она не выполняет поиск элемента справочника, а лишь фиксирует *план поиска* в виде *основы* (справочника *с*) и последовательности *слогов* (в примере – одного слога "ф1"). План поиска можно запомнить:

в := *с*//*ф1*

и затем использовать в операциях над файлами:

фв := *файл* (*в*)

Поиск по заданному плану будет выполняться только при исполнении операции *файл*, поскольку в ней требуется доступ к файлу. Подчеркнем, что внешнее имя позволяет абстрагироваться от физической ссылки на файл и в разные моменты времени может обозначать разные файлы (в рассматриваемом примере – файл *ф1* или файл *ф2*).

Базовый справочник контейнера. При создании контейнера на диске вместе с ним создается и его *базовый справочник*. Он служит для ссылок на создаваемые в контейнере объекты с помощью внешних имен. В качестве ссылки на базовый справочник используется внешнее имя контейнера, например, *ксн*//*лгу*. Следует учесть, что само по себе создание файла в контейнере не обеспечивает автоматического занесения его имени в справочник; эти действия должны быть запрограммированы явно, например:

создэлспр (*ксн*//*лгу*, " текст ", *генфайл* (*ксн*//*лгу*))

Здесь операция *генфайл* создает файл, *создэлспр* – создает его имя ("текст") в базовом справочнике. Созданный файл доступен с помощью внешнего имени *ксн*//*лгу*//*текст*. Аналогично в контейнере *ксн*//*лгу* можно создать еще один справочник (например, личный справочник файлов какого-либо программиста) и организовать ссылку через него на новый файл:

создэлспр (*ксн*//*лгу*, " иванов ", *генспр* (*ксн*//*лгу*));

создэлспр (*ксн*//*лгу*//*иванов*, " текст1 ", *генфайл* (*ксн*//*лгу*))

Внешнее имя нового справочника – *ксн//лгу//иванов*, нового файла – *ксн//лгу//иванов//текст1*.

При практической работе на МВК "Эльбрус" для создания файлов и справочников чаще всего используются более лаконичные конструкции языка стандартного диалога – расширения Эль-76 (п. 1.33).

Виды стандартной основы внешних имен.

1. *ксн* (контекст съемных носителей). *ксн* – справочник, составленный из имен всех контейнеров, которые смонтированы на внешних устройствах (дисках и лентах) в данный момент. Таким образом, если монтируется диск с контейнером, имеющим имя "итм", то этот контейнер автоматически получает внешнее имя *ксн//итм*. Если же при обращении по этому внешнему имени система обнаруживает, что данный контейнер отсутствует, то она выдает сообщение оператору о необходимости его установки. В случае, если установлены два контейнера с одинаковым именем, система при каждом открытии одного из них будет требовать уточнения (в виде номера устройства).

2. *свойвк* (внешний контекст программы – "свой внешний контекст"). Каждая программа имеет внешний контекст в виде корневого справочника, определяющего основу для используемых в ней внешних имен. Внешний контекст программы может быть задан компилятором или самим программистом. Внешние имена, построенные от внешнего контекста программы, имеют вид

свойвк//a1//...//ak

или

//a1//...//ak

(основа *свойвк* подразумевается по умолчанию).

Каждый пользователь, зарегистрированный в системе, также имеет свой внешний контекст, ссылка на который хранится в его *досье*. Этот контекст определяется администратором и автоматически устанавливается при входе в диалог с данным именем пользователя.

3. *глобарх* (глобальный архив) – это барабанный справочник, содержащий ссылки на файлы объектного кода всех системных программ ОСПО.

Например, *глобарх//печкода* – внешнее имя стандартной процедуры печати файла объектного кода.

1.23.3. Ликвидация элемента справочника. Элемент справочника ликвидируется операцией *ликвидэлспр* (п. 1.20.4):

ликвидэлспр (ксн//лгу//иванов//текст1)

В справочнике *ксн//лгу//иванов* ликвидируется элемент "текст1", и если на именуемый им файл нет других ссылок, то ликвидируется и сам файл.

Пример.

Процедура-функция, анализирующая внешнее имя, которое задано в виде байтовой строки, и выдающая ссылку на именуемый им объект. В случае ошибки результат – **пусто64**.

```
процедура дайвнешимя = функция (с)
до ошибка, * границамассива
    (ф64 стр = с,        % текущий указатель строки
         объект,         % ссылка на объект
         слог,           % указатель на символы слога
         длслг;          % длина слога
   % — — — определение основы внешнего имени — — — %
   если стр [ :3] @= "ксн"
   то   объект := ксн; стр := стр [3:]
   инес стр [:2] @= "//"
   то   объект := свойвк
   инес стр [:6] @= "свойвк"
   то   объект := свойвк; стр := стр [6:]
   инес стр [:7] @ = "глобарх"
   то   объект := глобарх; стр := стр [7:]
   иначе ошибка!
   все;
   % создание указателя и его настройка на справочник
   объект := обкт (генув ( ), объект);
   % — — — анализ слогов внешнего имени — — — %
   до * ситнетварх, конец цикл
        если длина стр = 0 то конец!
        инес стр [:2] @ = "//"
        то % — — — анализ текущего слова — — — %
             стр := стр [2:];
             слог := стр [ : длслг := длина стр –
                          длина (стр от не среди буквицифр)];
             объект := элспр (генув ( ), объект, слог);
             стр := стр [ длслг :]
         иначе ошибка!
         все
   повторить
   при  ситнетварх: ошибка!
   всесит;
   объект)
при ошибка: , границамассива: пусто64
всесит % — — — дайвнешимя — — — %
```

1.23.4. Операции анализа справочника. В практике программирования сложился стиль работы со справочниками (*библиотеками*), основанный на выполнении указанных действий (трансляции, распечатки и т.п.) над некоторым подмножеством элементов библиотеки. Выбор этого подмножества, как правило, определяется именами элементов (например, их префиксами), а также значениями атрибутов.

Для *просмотра* справочника (от начала к концу или от конца к началу) служат операции *начспр*, *следэлспр*, *конспр* и *предэлспр*. В этих операциях (как и во всех действиях над справочниками) для настройки на элемент используется подвижный указатель *ув*. Операция *насчпр* (*ув*, *спр*) устанавливает указатель *ув* в позицию перед начальным элементом справочника *спр*. Операция *следэлспр* (*ув*) передвигает указатель на следующий элемент

справочника. Аналогичный смысл имеют операции *конспр* (*ув*, *спр*) и *предэлспр* (*ув*), осуществляющие просмотр справочника в обратном направлении. При просмотре справочника возникает ситуация *ситнетвархх*, если элементы справочника в данном направлении исчерпаны.

Указатель *ув* можно использовать для анализа внешнего объекта, именуемого через элемент справочника. Однако следует учитывать, что элемент может не ссылаться ни на какой объект, т.е. содержать пустую внешнюю ссылку *пустовнеш*. В этом случае при попытке обращения к объекту возникает ситуация *ситнетархсслк* (нет архивной ссылки).

Имя текущего элемента справочника является значением атрибута *имяэлспр*. Для его получения (как и в случае имени контейнера, п. 1.22) требуется вспомогательная байтовая строка *с*:

читатр (*ув*, *с*, *имяэлспр*) –

указатель на строку символов, образующих имя элемента справочника.

Пример.

Сервисная процедура анализа справочника, выполняющая заданные действия над указанным подмножеством тех его элементов, которые являются текстовыми файлами (*типфайла* > =3). Подмножество элементов задается характеристической функцией *принадлежит* (*имя*), где *имя* – указатель на символы имени элемента. Действия над выбранным элементом задаются в виде процедуры *обработать* (*ув*), где *ув* – указатель на элемент.

```
процедура анализспр = проц (спр, принадлежит, обработать)
 (ф64  ув := генув (),                  % подвижный указатель
       буф := лок вект [20] ф8,         % буфер имени элемента
       имя,                             % указатель имени элемента
       печ := читатрзадачи (виртвыв), % файл для печати протокола
       чисэл := 0,                      % общее число элементов
       чисэлобр := 0;                   % число обработанных элементов
 до    * ситнетвархх
       (запф (печ, :"– – – начало обработки справочника – – – " l);
        начспр (ув, спр);
        цикл  % завершается по ситнетвархх
              следэлспр (ув);
              чисэл := * + 1;
              если    % проверка элемента на текстовый файл
                      до * ситнетархсслк, * ситошатр
                         (читатр (ув, типфайла) >= 3)
                      при ситнетархсслк:, ситошатр: ложь
                      всесит
              то      % элемент – текстовый файл
                      имя := читатр (ув, буф, имяэлспр);
                      если принадлежит (имя)
                      то  % обработка элемента
                          чисэлобр := * + 1;
                          запф(печ, стр8"обработка элемента:",
                               имя: п (длина имя) а, : l);
                          обработать (ув)
```

```
                    все
              все
        повторить)
   при ситнетварх:
        запф (печ, чисэл: "всего элементов="g (10),
                  чисэлобр: "обработано элементов="g (10),
                  : " – – – конец обработки справочника – – – " l)
   всесит) % – – – анализспр – – – %
```

Пример использования:

```
анализспр (ксн//лгу
          функция (имя)
             если длина имя < 3 то ложь
             иначе имя [:3] @= "саф"
             все,
          проц (ув) (печтекстфайл (ув)))
```

В этом примере процедура *анализспр* используется для распечатки всех текстовых файлов справочника *ксн//лгу*, имена которых имеют префикс "саф". *печтекстфайл* – системная процедура распечатки текстового файла.

Поясним избранный метод проверки на текстовый файл (см. процедуру *анализспр*). При исполнении операции *чит атр* (*ув, типфайла*) могут возникнуть ситуации: *ситнетархсылк* – если элемент ни на что не ссылается – и *ситошатр* (ошибка атрибута) – если элемент ссылается не на файл. Возникновение одной из этих ситуаций означает, что элемент справочника не ссылается на текстовый файл.

1.23.5. Справочник внешних связей файла (СВС). СВС предназначен для организации ссылок между файлами. Некоторые из этих ссылок носят стандартный характер, устанавливаются компиляторами и используются программами ОСПО. Однако допускаются и другие внешние ссылки между файлами, например, для организации баз данных.

СВС хранится в заголовке файла. Число элементов СВС задается значением атрибута *длинсвс* (по системному умолчанию – 0), которое может быть увеличено. Например:

```
ф := генфайл (барабан, длинсвс: 64); . . .
запатр (ф, длинсвс: 128)
```

Над СВС определены те же операции, что и над обычным архивным справочником. В качестве ссылки на СВС выступает ссылка на файл, в качестве имени элемента в СВС – его номер (элементы СВС нумеруются начиная от нуля). Например, операция

```
создэлспр (ф, 0, ф1)
```

создает в СВС файла *ф* элемент с номером 0 и записывает в него ссылку на файл *ф1*.

Рассмотрим организацию стандартных внешних связей файлов через СВС.

1. *Связь файла объектного кода (ФОК) с его дополнением (ДФОК).* Файл объектного кода имеет атрибут *инфпрогр*, значением которого является индекс элемента СВС, содержащего ссылку на ДФОК. Эту ссылку

устанавливает компилятор при трансляции программы

```
запатр (фок, инфпрогр: 0) ;
создэлспр (фок, 0, дфок)
```

2. *Связь ФОК с исходным текстом программы* устанавливается аналогичным образом через атрибут *текстпрогр.*

3. *Связь файла с его внешним контекстом* задается так же, как в двух предыдущих случаях, через значение атрибута *внешконт.* Для *текстового файла,* в котором хранится фрагмент текста программы, атрибут *внешконт* задает ссылку на ФОК программы, в контексте которой должны пониматься все используемые глобальные идентификаторы. Для *файла объектного кода* этот атрибут обозначает ссылку на корневой справочник внешнего контекста программы (по умолчанию – *глобарх*).

Обращение к элементу СВС через внешнее имя. Для элементов СВС предусмотрена особая форма внешних имен: *ф*// [*к*], где *ф* – ссылка на файл, *к* – индекс элемента в СВС. Например, *фок*// [0] – внешнее имя ДФОК.

1.23.6. Условия применения архивных операций.

1. В элемент справочника (архивного или СВС) можно записать ссылку на внешний объект, находящийся *в одном контейнере* со справочником. Более точно, это означает следующее. Если справочник барабанный, то он может содержать только ссылки на барабанные объекты, дисковый – только ссылки на объекты, расположенные в том же дисковом контейнере. Ссылки из справочников на другие носители (файлы на ленте, на дисплее и т.п.) не допускаются.

2. Длина имени элемента справочника не должна превышать 17 символов.

В перспективе предполагается введение в систему простых файлов традиционного понятия *каталогизации*, при котором возможны будут ссылки из одного каталогизированного контейнера в другой.

Пример.

Пусть *фок* – файл объектного кода. Требуется создать файл для ДФОК в одном контейнере с ФОК. В компиляторах выполнение этой рутинной задачи берет на себя технологический пакет ИНТЕРФОК.

```
(ф64   буф := лок вект [20] ф8,   % буфер для имени контейнера
       конт;                       % имя контейнера
если    читатр (фок, типву) = 1   % барабан
то      дфок := генфайл (барабан)
иначе   % диск (для простоты не проверяется)
        конт := элспр (генув (), ксн, читатр (фок, буф, имяконт)) ;
        дфок := генфайл (конт)
все ;
запатр (фок, инфпрогр: 0) ;
до * ситошпарамарх
        (создэлспр (фок, 0, дфок))
при ситошпарамарх: запспр (элспр (генув (), фок, 0), дфок)
всесит)
```

Поясним использование архивных операций. Ссылка на дисковый контейнер вычисляется с помощью атрибута *имяконт*, значением которого (для дисковых файлов) является символьное имя контейнера. В операции *элспр* идентификатор *ксн* используется как обычный справочник. При связывании *фок* и *дфок* может возникнуть ситуация *ситошпарамарх* в операции *создэлспр*, если *фок* уже имеет нулевой элемент СВС. В этом случае используется операция *запспр.*

1.24. Обмен с барабанами и дисками

Виды и способы обмена были кратко рассмотрены в п. 1.20.3. Способы и операции обмена в некоторой степени зависят от типа внешнего устройства, хотя в обозначении операций и достигнута максимально возможная степень унификации. Поэтому операции обмена и специфические атрибуты рассматриваются для каждого типа устройств (или группы родственных типов устройств) отдельно (пп. 1.24–1.26).

1.24.1. Особенности организации и атрибуты файлов. Несмотря на различие в физической организации, магнитные диски и барабаны в СПФ унифицированы, так как являются *устройствами произвольного доступа.* Минимальным квантом обмена с барабаном и диском является *сектор* – 32 слова. Память файла на диске или барабане логически делится на секторы, нумеруемые от нуля.

Выделение памяти под файл на диске или барабане осуществляется в более крупных единицах – *листах.* Длина листа задается значением атрибута *длинлиста* при генерации файла и измеряется в секторах (по системному умолчанию – 128 сектора). Длина листа файла, в который произведена хотя бы одна запись, не может быть изменена. По аналогии с массивами, при генерации файла создается лишь его заголовок. Память под очередной лист файла выделяется только при первой записи в сектор, входящий в этот лист. При попытке чтения из несуществующего листа возникает ситуация *ситнетлиста.* Максимальная длина файла в листах задается значением атрибута *максдлинфайла* (по умолчанию – 10 листов).

Особенности хранения информации. На магнитных барабанах информация может храниться вместе *с тегами.* На дисках также можно хранить информацию с тегами: для них существует стандартный способ упаковки, при котором в 32 словах дискового сектора размещается 28 слов с тегами. Способ хранения информации определяется значением атрибута *типданных* (**истина** – с тегами, **ложь** – без тегов). Для файла на диске возможны оба способа хранения. Для файла на барабане значение этого атрибута всегда **истина**. При истинном значении атрибута *типданных* упаковка и распаковка тегов при обмене с диском обеспечивается системой, однако программист должен учитывать фактический размер информации (например, массив в 60 слов с тегами помещается на диске лишь в 3 сектора, а на барабане – в 2 сектора). Если значение атрибута *типданных* равно нулю, то при чтении данных им приписывается тег "набор".

Попытка записи в файл адресной и управляющей информации (указателей, меток и т.п.) с тегами в непривилегированном режиме приводит к стандартной ситуации *ситпривобмен* (самой записи не происходит).

1.24.2. Непосредственный обмен возможен в двух вариантах: *синхронный* (программа ждет окончания обмена) и *параллельный* (параллельно с выполнением программы). Для дисков и барабанов рассматривается только первый вариант, как более предпочтительный.

Для непосредственного синхронного обмена файл открывается следующим образом:

з := *файл* (*ф*)

где *ф* – ссылка на файл в любой из трех форм (пп. 1.21, 1.23). Переменная *з* обозначает заголовок открытого файла.

1. *Операция записи*:

зап (*з, нсект, в*)

Здесь *нсект* – номер сектора, с начала которого выполняется запись; *в* – указатель вектора, содержимое которого записывается в файл. Запись с произвольного слова сектора невозможна. Если длина вектора *в* не кратна сектору, то остаток последнего сектора заполняется нулями. При выполнении операции *зап* возможны следующие ситуации:

ситфкф – при превышении *максдлинфайла*;

ситошпарамобмена – в случае неверных параметров операции и в случае, если записываемый вектор не помещается в один лист файла.

Ввиду последней особенности пользоваться непосредственным обменом рекомендуется лишь для сравнительно небольших векторов. При буферизованном обмене превышение длины листа при записи в файл не приводит к прерыванию; эта ситуация обрабатывается системой.

2. *Операция чтения*:

чит (*з, нсект, в*)

Параметры имеют тот же смысл, что и в операции *зап*. Из файла, начиная с начала сектора *нсект*, считывается в вектор *в* число элементов, равное его длине. При попытке чтения из листа, в который еще не было записи, возникает ситуация *ситнетлиста*.

Закрытие файла по окончании всех операций обмена должно быть выполнено операцией:

откреп (*з*)

П р и м е р ы.

1. *зап* (*фок*, 0, *описатель*) – запись описателя файла объектного кода при завершении его генерации (описатель занимает нулевой сектор ФОК).

2. *чит* (*фок*, 0, @ *описатель* [0]) – чтение из файла объектного кода нулевого слова описателя ФОК, содержащего дату его создания.

В практике программирования для МВК "Эльбрус" непосредственный обмен с диском и барабаном применяется сравнительно редко, так как он не обеспечивает обмен с точностью до слова и не отслеживает логический конец файла.

1.24.3. Буферизованный обмен. При буферизованном обмене файл логически разделен на *блоки*. Для файла на диске или барабане все блоки имеют постоянную длину, равную значению атрибута *максдлинблока* (по системному умолчанию – 256 слов, т.е. 8 секторов). Число заполненных блоков файла определяется значением атрибута *блоклкф* (блок логического

конца файла. Рассмотрим наиболее простой способ буферизованного обмена – *однобуферный обмен,* при котором в каждый момент доступен *один блок* файла.

При открытии файла для однобуферного обмена создается позиционная переменная:

п := *позп* (*ф*)

Позиционная переменная задает текущую позицию по файлу и может устанавливаться на любой блок. По умолчанию она установлена в позицию перед начальным блоком файла. Возможны два способа доступа к файлу – *последовательный* (относительно текущего блока) и *произвольный* (по номеру блока).

Последовательный доступ. Операции чтения и записи имеют вид: *чит* (*п, след*), *зап* (*п, след*), *нов* (*п, след*). Стандартный идентификатор *след* задает направление движения по файлу – *к следующему блоку.* Идентификатор *пред* (вместо *след*) задает перемещение *к предыдущему блоку.*

Операции чтения и записи при буферизованном обмене делают содержимое текущего блока доступным через *указатель буфера,* который является результатом операции. Операция *чит* выдает буфер, защищенный по записи, операция *зап* – буфер, содержимое которого можно менять. В операциях *чит* и *зап* в буфер считывается первоначальное содержимое блока, в операции *нов* этого не происходит (она предназначена для полного обновления блока). Формат буфера – **ф64.** Если более удобен другой формат, рекомендуется установить его при создании позиционной переменной с помощью атрибута *формэлем* (формат элемента). Например, если

п := *позп* (*ф, формэлем*: 3)

то формат буферов устанавливается равным **ф8.** При перестановке позиционной переменной на другой блок система автоматически сбрасывает содержимое буфера в текущий блок (если над ним была выполнена операция *зап* или *нов*).

Пример.

Процедура переписи барабанных или дисковых файлов.

```
процедура переписьмбмд = проц (ф1, ф2)
до * ситфкф
 (ф64 п1,              % позп на файл ф1
     п2,               % позп на файл ф2
     чисблок := читатр (ф1, блоклкф), % число блоков ф1
     длинблок := читатр (ф1, максдлинблока); % длина блока ф1
 запатр (ф2, максдлинблока: длинблок);
 п1 := позп (ф1);
 п2 := позп (ф2);
 от 1  до чисблок цикл
     нов (п2, след) < := ф64 чит (п1, след)
 повторить;
 откреп (п1); откреп (п2)
 при ситфкф:
   запф (читатрзадачи (виртвыв),
     стр8 "переполнение!", : l)
 всесит  % – – – переписьмбмд – – – %
```

В примере операции *чит* и *нов* последовательно выдают указатели буферов на блоки 0, 1, 2, ... файлов *ф1* и *ф2*. Эти указатели используются в операции пересылки как обычные указатели векторов. Перед началом обмена длина блока *ф2* устанавливается равной длине блока *ф1*. Закрытие файлов здесь обязательно, так как при этом обеспечивается запись последнего блока.

При последовательном доступе возможен сдвиг более чем на 1 блок. Например, операция *чит* (*п*, *след* (10)) сдвигает позиционную переменную *п* на 10 блоков вперед.

Произвольный доступ. Операции обмена записываются в несколько иной форме:

чит (*п*, *нблок*), *зап* (*п*, *нблок*), *нов* (*п*, *нблок*)

где *нблок* – номер блока. Операции настраивают позиционную переменную *п* на блок с номером *нблок*. В остальном они выполняются точно так же, как и при последовательном доступе. Оба способа доступа могут комбинироваться. Одна и та же позиционная переменная может использоваться и для чтения, и для записи. Более того, с одним файлом на диске или барабане могут быть одновременно связаны несколько позиционных переменных (что служит адекватной заменой более сложного многобуферного способа обмена).

Пример.

Процедура чтения вектора из файла.

```
процедура читвект = функция (ф, номслова, длина);
% чтение содержимого файла ф, начиная со слова с заданным
% номером, заданной длины. Результат – указатель глобально-
% го вектора; при выходе за границу файла – пусто64
до  * ситлкф
    (ф64 п := позп (ф),                 % позп на файл
         вектор := глоб вект [длина] ф64,  % вектор-результат
         теквект := вектор,             % текущий указатель вектора
         длинблок := читатр (ф, максдлинблока), % длина блока
         текблок := чит (п, номслова /: длинблок)
                          [номслова остат длинблок: ];
                                      % текущий указатель блока
    до   концевект цикл
         мод теквект <:= ф64 мод текблок;
         если ттп % вектор заполнен
         то   концевект!
         иначе % переход к следующему блоку
              текблок := чит (п, след)
         все
    повторить;
    вектор)
при ситлкф:
    запф (читатрзадачи (виртвыв), : "чтение за границей файла" l);
    пусто64
всесит % – – – читвект – – – %
```

1.25. Обмен с лентами

В этом параграфе описываются средства и особенности обмена с магнитной лентой. Как и на других типах ЭВМ, ленты на МВК "Эльбрус" используются в основном для сохранения и переноса информации. В составе ОСПО имеются стандартные программы, обеспечивающие:

– чтение на МВК "Эльбрус" лент, записанных различными способами на ЕС и СМ ЭВМ;

– запись на ленту совокупности файлов из одного или нескольких справочников и их перепись с ленты на диск;

– Запись на ленту содержимого дискового контейнера и восстановление его с ленты.

Данный параграф предназначен в основном для системных программистов.

1.25.1. Структура и атрибуты контейнера на лентах. В системе "Эльбрус" используются стандартные накопители на магнитных лентах ЕС ЭВМ. Внутренняя структура контейнера на лентах аналогична структуре ленты в ОС ЕС ЭВМ (хотя, по сравнению с последней, в метках хранятся некоторые дополнительные атрибуты). Контейнер на лентах имеет следующую структуру:

$$нк\ ф_1\ кф_1\ нф_2\ ф_2\ кф_2\ \ldots\ нф_n\ ф_n\ кф_n\ кк,$$

где *нк* – начальная метка контейнера, совпадающая с начальной меткой первого файла; $нф_i$ – начальная метка i-го файла; $ф_i$ – содержимое i-го файла; $кф_i$ – конечная метка i-го файла; *кк* – конечная метка контейнера. Если контейнер занимает несколько лент (*томов*), то каждый том имеет аналогичную структуру. Файлы в контейнере нумеруются начиная с единицы.

Контейнер на ленте, как правило, имеет *имя* (идентификатор), записываемое в его начальную метку при создании (разметке) контейнера:

генконтейнер (*млконт*, *имяконт*: "эталон")

В этом случае контейнер называется *помеченным.* Имя контейнера является значением атрибута *имяконт.* В большинстве случаев используются помеченные контейнеры, поэтому в дальнейшем будем рассматривать только их.

Все файлы помеченного контейнера на лентах являются также помеченными и имеют свои имена. Имя файла на ленте хранится в его начальной метке и задается с помощью атрибута *имяфайла* (п. 1.25.2).

Атрибут контейнера *файлконт* предназначен для установки текущей позиции ленты на файл с заданным номером (п. 1.25.2). Имеются также атрибуты контейнера для управления томами (если их несколько).

1.25.2. Непосредственный обмен. При обмене с лентой необходимо учитывать следующие особенности:

1. Поскольку к ленте возможен только последовательный доступ, перед созданием или обработкой файла необходимо *установить текущую позицию* ленты на этот файл или на точку его создания.

2. Не рекомендуется хранить на ленте в одном файле записи разной длины. Чаще всего запись на ленту выполняется либо "картами" по 80 байтов

(текст в формате ЕС ЭВМ), либо дисковыми блоками по 2048 байтов (произвольная информация), хотя возможна и другая длина записи.

3. Доступ к ленте (в отличие от диска и барабана) *монопольный*, поэтому после завершения обмена необходимо выполнить закрытие файла и контейнера. В противном случае (либо при сбое устройства) лента будет считаться занятой.

Открытие контейнера выполняется операцией *контейнер* (1.22). По умолчанию лента устанавливается на начало. Атрибут *файлконт* позволяет установить начальную позицию ленты на определенный файл:

лк := *контейнер* (*ксн*//*эталон*, *файлконт*: 5)

Контейнер *лк* устанавливается на начало файла номер 5.

Создание файла на ленте выполняется всегда *в текущей позиции*, в которую она установлена:

ф := *генфайл* (*лк*, *имяфайла*: "текст1")

Создается файл с именем "текст1" и номером 5. Для обмена с файлом необходимо *открытие файла*:

фоткр := *файл* (*ф*)

Операции обмена. Если *в* – вектор (строка) байтового формата, то операция

зап (*фоткр*, *след*, *в*)

создает новую *запись* файла в текущей позиции и заполняет ее значениями элементов вектора *в*. При этом значение атрибута файла *блокфайла* устанавливается равным текущему *номеру записи* в файле.

Чтение текущей записи из файла в байтовый вектор *в* выполняется операцией:

чит (*фоткр*, *след*, *в*)

Если длина записи больше длины вектора, то значение атрибута файла *переполнен* становится равным 1.

Пример.

Запись на заданную ленту барабанного или дискового файла под заданным именем и с заданным номером на ленте. Файл записывается блоками.

```
процедура сброснамл = проц (имяленты, номерф, имяф, ф)
(ф64  л := контейнер (элспр(генув(), ксн, имяленты), файлконт:
                                                        номерф),
      позф := позп (ф, формэлем: 3),
      фл := генфайл (л, имяфайла, имяф),
      фленты := файл (фл),
      чисблокф := читатр (ф, блоклкф),
      блокф;
 от 1 до чисблокф цикл
      блокф := чит (позф, след); % формат буфера – ф8
      зап (фленты, след, блокф)
 повторить ;
 откреп (позф); откреп (фленты); откреп (л)
) % – – – сброснамл – – – %
```

Установка ленты. Операция *уст* (*фоткр, нач*) устанавливает ленту в позицию непосредственно за начальной меткой файла *фоткр*; *уст* (*фоткр, след*) – пропускает одну запись файла, *уст* (*фоткр, след* (*к*)) – *к* записей файла.

Пример.

На заданной ленте под заданным номером записан файл. Требуется переписать в дисковый файл *ф* содержимое файла на ленте начиная с 100-й записи.

```
л := контейнер (элспр(генув (), кнм, "эталон"), файлконт: ном);
п := позп (ф, формэлем: 3);
фл := файл (л);
уст (фл, след (100));
блок := нов (п, след);
до * ситлкф цикл
    чит (фл, след, блок); блок := нов (п, след)
повторить;
откреп (п); откреп (фл); откреп (л)
```

1.25.3. Буферизованный обмен с лентой более удобен по форме выражения, чем непосредственный. Однако по семантике он почти не отличается от непосредственного, так как *блоком* на ленте считается одна *запись* файла. Ввиду последовательной организации доступа к ленте имеются лишь следующие разновидности операций буферизованного обмена (ср. п.1.24.3):

чит (*п, след*), *зап* (*п, след*), *нов* (*п, след*), *уст* (*п, след*)

и их варианты со сдвигом на *к* блоков (*след* (*к*)) и с реверсивной обработкой (*след* → *пред*). По умолчанию формат буфера блока для лент равен **ф8**.

Пример.

Решение задачи записи файла на ленту (п. 1.25.2) с помощью буферизованного обмена.

```
процедура записьнамл = проц (имяленты, номерф, имяф, ф)
(ф64
    л := контейнер (элспр(генув (), ксн, имяленты), файлконт:
                                                    номерф),
    позф := позп (ф, формэлем: 3),
    фл := генфайл (л, имяфайла: имяф),
    позл := позп (фл),
    чисблокф := читатр (ф, блоклкф);
от 1 до чисблокф цикл
   нов (позл, след) <:= чит (позф, след)
повторить;
откреп (позф); откреп (позл); откреп (л)
), % – – – записьнамл – – – %
```

1.25.4. Параллельный обмен. Для магнитных лент имеет смысл применять наряду с синхронным способом обмена параллельный, при котором обмен выполняется параллельно с программой, а ожидание окончания обмена осуществляется с помощью семафора. Для параллельного обмена ис-

пользуется блок ввода-вывода (бвв):

л := *контейнер* (*элспр*(*генув* (), *ксн*, "эталон"), *файлконт*: 2);
фбвв := *бвв* (*л*),
устпар (*фбвв, след* (1000)); % перемотка ленты
% выполнение основной программы
ждать (*читатр* (*фбвв, смфобмена*)); % ожидание
% этот фрагмент исполняется после установки ленты

Здесь *смфобмена* – атрибут *бвв*, значением которого является *семафор обмена*. В примере он используется для синхронизации обмена с основной программой. Имеется "параллельная" форма и других операций обмена с лентой:

читпар (*фбвв, след, стр*), *заппар* (*фбвв, след, стр*) и т.д.

1.26. Терминальные файлы. Диалог

С помощью терминальных файлов, связанных с алфавитно-цифровыми дисплеями (*ацд*), в системе "Эльбрус" организуется диалоговый режим исполнения программ. Как уже отмечалось (п. 1.21.1), терминальный файл связывается с определенной диалоговой задачей и создается системой при входе пользователя с терминала в стандартный диалог. В языке стандартного диалога (п. 1.33) терминальный файл задачи обозначается э или *мойтерм*; программа может получить доступ к нему через фактический параметр при запуске программы с терминала:

исп//*код* (э),

где //*код* – внешнее имя кода программы.

1.26.1. Структура и атрибуты терминального файла. Терминальный файл состоит из *сообщений*. Сообщения делятся на *входные* (набираемые программистом и вводимые программой с терминала) и *выходные* (выводимые на терминал). Сообщения могут иметь различную длину: от 0 (*пустая посылка*) до размера экрана. На МВК "Эльбрус" используются в основном терминалы ЕС–7920. В первой позиции каждой строчки экрана система проставляет управляющий символ, так что размер экрана в СПФ – 24 строки по 79 символов.

Система простых файлов обеспечивает лишь самую простую форму диалога – ввод-вывод сообщений начиная с текущей позиции курсора. Более удобная форма диалога – *многооконный обмен* – реализована в системе структурированных файлов.

Диалог может вестись *с приглашением* для ввода входного сообщения. Символы приглашения считаются частью сообщения. Форма приглашения определяется значением *п* атрибута *приглашение* терминального файла:

1. если *п* = 0, диалог осуществляется *без приглашения* (признаком начала ввода сообщения является установка курсора на начало следующей строки);.

2. при *п* = 1 выдается *приглашение в виде номера строки*. Этот вид приглашения используется в текстовом редакторе (п. 1.33.7) при вводе новых строк текста и в трансляторах – при вводе исходного текста. Начальный номер строки задается значением атрибута *фнс* (фиксированный

номер строки), шаг нумерации – значением атрибута *шагфнс*. Номер состоит из 8 цифр;

3. при $n > 1$ значение n задает код символа, который выдается в качестве приглашения. Например, n = ">" – стандартное приглашение в виде символа ">".

1.26.2. Непосредственный обмен. Будем предполагать, что *экран* – ссылка на терминальный файл, получаемая в качестве параметра при запуске программы с терминала:

программа = **проц** (*экран*, . . .)

В процессе работы диалоговой программы должны выполняться следующие действия по организации обмена с экраном:

1. Сохранение приглашения (если диалог будет вестись с нестандартным приглашением):

сприглашение := *читатр* (*экран, приглашение*)

2. Открытие терминального файла:

фэкран := *файл* (*экран*)

3. Установка приглашения. Например, приглашение в виде номера строки, начиная с нуля с шагом 1000, устанавливает операция:

запатр (*фэкран, приглашение*: 1, *фнс*: 0, *шагфнс*: 1000)

Приглашение в виде символа "=" задается следующей операцией:

запатр (*фэкран, приглашение*: "=")

4. Установка курсора на нужную строчку экрана. Для управления курсором служит атрибут *строчка*. Операция

запатр (*фэкран, строчка*: 1)

устанавливает курсор на первую строчку экрана.

5. Вывод сообщений на терминал выполняется операцией записи:

зап (*фэкран, след, сообщение*),

где *сообщение* – указатель байтовой строки. Сообщение может быть любой длины; перевод строчки экрана осуществляется автоматически. Однако чаще всего сообщение имеет длину не более строчки (79 символов). Каждое новое сообщение выводится с новой строчки. Например, последовательность операций:

зап (*фэкран, след,* **стр8** "транслятор паскаль–эльбрус");
зап (*фэкран, след,* **стр8** "версия 1.09.87")

выводит на экран две указанные строки. Операция:

зап (*фэкран, след, чистэкран*)

где **конст** *чистэкран* = **мконст ф8** (/3 * 79/ "   ") , очищает экран пробелами. Отметим следующие особенности:

– обмен с терминалом возможен только *символьной информацией* (для ввода-вывода чисел рекомендуется пользоваться форматным обменом п. 1.19);

– терминал в СПФ считается устройством последовательного доступа (хотя текущую позицию можно изменить с помощью атрибута *строчка*).

6. Ввод сообщений с терминала осуществляется операцией *чит*:

чит (*фэкран, след, стр*)

где *стр* – указатель байтовой строки для записи сообщения. Сообщения записываются в строку *стр* вместе с приглашением. Конец сообщения определяется нажатием клавиши "ввод". Фактическая длина сообщения равна значению атрибута *длинблока*. Например, если было выдано приглашение в виде номера строки 00001000, а программист на экране набрал символы *begin* и нажал клавишу "ввод", то значение атрибута *читатр* (*фэкран, длинблока*) равно 13, а содержимое строки равно:

00001000 *begin*

Если сообщение представляет собой пустую посылку, то

читатр (*фэкран, длинблока*) = 8

а строка *стр* содержит только номер строки. При стандартном приглашении ">" для тех же двух сообщений ("*begin*" и пустой посылки):

– *читатр* (*фэкран, длинблока*) = 6: содержимое *стр*: > *begin*;
– *читатр* (*фэкран, длинблока*) = 1, содержимое *стр*: >.

7. Указание конца терминального файла. В СПФ терминальный файл считается потенциально бесконечным; стандартных средств для указания конца файла не предусмотрено. Рекомендуются два способа:

а) пустая посылка (распознается программой по значению атрибута *длинблока*);

б) подмена символа приглашения на экране (например, набор вместо стандартного приглашения ">" символа "#"). При этом в буфер сообщения попадает, вместо приглашения, набранный на экране символ, что легко распознается программой.

Первый способ более удобен, если приглашение выдается в виде номера строки – например, при наборе с экрана текста программы для трансляции. Второй способ предпочтителен при выдаче приглашения в виде символа. Следует учитывать, что программист, недостаточно знакомый с диалоговой системой, склонен часто нажимать клавишу "ввод", т.е. использовать пустую посылку, поэтому в качестве признака конца файла она неудобна.

8. Учет заполнения экрана. При "построчном" стиле диалога и большом объеме выдачи на терминал весьма неудобно просматривать выходную информацию. СПФ не имеет стандартных средств задержки вывода при заполнении экрана. Однако с помощью переменной-счетчика и атрибута *приглашение* они легко реализуются: при вводе и выводе каждой новой строки счетчик увеличивается на единицу; если счетчик равен размеру экрана, то в качестве приглашения выдается символ "?" (пустая посылка – продолжение вывода, любая другая – окончание).

9. Закрытие терминального файла:
откреп (*фэкран*)

10. Восстановление приглашения:
запатр (*экран, приглашение*: *сприглашение*)

Пример.

Диалоговая программа выдачи информации в файлах. Внешние имена файлов вводятся с терминала. Приглашение – символ "=", признак конца диалога – набор символа "к" вместо приглашения.

```
процедура инфайл = проц (э)
начало
  ф64 сприглашение := читатр (э, приглашение),
      экран := файл (э),
      буф := лок вект [80] ф8,          % буфер для сообщений
      сообщ,                            % указатель сообщения
      укфайл,                           % ссылка на текущий файл
      счетстрок := 0;                   % счетчик строчек экрана
статсит конецработы;
процедура конецэкрана;  % предварительное описание
текст
    этофайл (ф) = % – – ф – файл? – – – %
      (читатр (ф, типвнешоб) = 2)),  % см. 1.29.1
    увеличьэкран = % – – увеличение счетчика строк на экране
      если (счетстрок := * + 1) = 23 то конецэкрана() все ,
    ввод = % – – ввод сообщения – – %
      (увеличьэкран;
      чит (экран, след, буф);
      если буф [1] = "к" то конецработы!; 0
      иначе буф [1: читатр (экран, длинблока) – 1]
      все ), % – – ввод – – %
    вывод (с) = % – – вывод сообщения с – – %
      (увеличьэкран;
      зап (экран, след, с)) ;  % – – вывод – – %
процедура конецэкрана = проц ( )
      (запатр (экран, приглашение: "?") ;
      чит (экран, след, буф);
      если буф [1] = "к" то конецработы!
      иначе  счетстрок := 0;
             запатр (экран, приглашение: "=")
      все ) ; % – – конецэкрана – – %
процедура дайвнешимя = функция (с)  ( . . . ) ;  % см. 1.23.3
% – – – начало работы системы – – – %
до  * ситошву  % до ошибки внешнего устройства
    (запатр (экран, приглашение: "=") ;
    вывод (стр8 "– – программа < информация о файле > – – ");
    до  * конецработы цикл
        вывод (стр8 "введите имя файла: ") ;
        сообщ := ввод;
        укфайл := дайвнешимя (сообщ) ;
        если естьпусто укфайл
        то вывод (стр8 "неверно задано имя файла") ;
        инес этофайл (укфайл)
```

```
            то % – – – формирование сообщения – – – %
               мод текбуф < := "файл:" &&"типву="&&
               если читатр (укфайл, типву) = 1 то "барабан"
               иначе "диск"
               все
               && "число"&&"блоков="
               && распак (дес читатр (укфайл, блоклкф))
               && "длина"&&"листа="
                   распак (дес читатр (укфайл, длинлиста));
               вывод (буф [: длина буф – длина текбуф]);
               текбуф := буф
            иначе % – – – не файл – – – %
                вывод (стр8 "заданный объект – не файл!")
            все
     повторить ;
     вывод (стр8 "– – – конец работы программы ‹ инфайл › – – – ");
     откреп (экран);
     запатр (э, приглашение: сприглашение)
   ) при ситошву: % сбой при работе программы
         вывод (стр8 "– – работа прекращена из-за сбоя – – ");
         откреп (экран);
         запатр (э, приглашение: сприглашение)
     всесит
     конец % – – – инфайл – – – %
```

1.26.3. Буферизованный обмен. Как и для магнитных лент, буферизованный обмен с терминалом имеет значительное сходство с непосредственным, так как *блоком* терминального файла считается одно *сообщение*. Как и при непосредственном обмене, атрибут *длинблока* задает длину прочитанного сообщения. Операции:

фэкран := файл (экран)

непосредственного обмена соответствует операция:

пэкран := позп (экран)

буферизованного обмена; операции

зап (фэкран, след, сообщ)

– операция:

зап (пэкран, след) < := сообщ

операции:

чит (фэкран, след, сообщ) –

операция:

сообщ < := чит (пэкран, след)

Все атрибуты (*строчка, приглашение* и т.п.) сохраняют смысл и для позиционной переменной *пэкран*. В отличие от лент, для терминалов форма *след*(*к*) ($к > 1$) не допускается.

1.27. Текстовые файлы

Текстовый файл – это файл стандартной структуры для хранения текстовой информации. Принадлежность файла *ф* к этому классу в системе простых файлов определяется значением атрибута *типфайла*: для текстовых файлов

читатр (*ф*, *типфайла*) > = 3.

СПФ не имеет операций над логической структурой текстового файла, а лишь фиксирует ее через набор атрибутов. Операции ввода-вывода над текстовыми файлами выполняются процедурами форматного обмена (п. 1.19). Средства работы с текстовыми файлами, удобные для системных задач, реализованы в технологическом пакете ИНТЕРТЕКСТ [8].

1.27.1. Структура и атрибуты текстового файла. Текстовый файл может создаваться на любом внешнем устройстве. Общей чертой его организации, не зависящей от типа устройства, является логическое разбиение на строки. Однако представление строк, их разделители и внутренняя структура для разных типов носителей различны. Это учтено в системе атрибутов текстового файла, определяющих длину строки и представление ее номера.

1. Атрибут *длинстрок* задает длину строки. Если он равен нулю, то строки имеют переменную длину, больше нуля – постоянную, равную значению атрибута. Первый случай характерен для дисков и барабанов, второй – для перфокарт (*длинстрок* = 80).

2. Атрибут *длинфнс* указывает длину номера строки. Как правило, строка текста хранится вместе с ее уникальным номером, т.е. текст является нумерованным. Если значение атрибута *длинфнс* равно нулю, то файл является не нумерованным. Атрибут *длинфнс* задает длину номера в символах (как правило, она равна 8). Номера строк в файле должны возрастать, иначе системные программы (например, редактор) зафиксируют ошибку в структуре файла.

3. Атрибут *кодфнс* описывает способ *хранения номера строки:* 0 – *двоичный* вид, 1 – *символьный* в коде ДКОИ. При хранении номера в двоичном виде он занимает минимально необходимое количество байтов для представление числа из *длинфнс* цифр (при *длинфнс* = 8 – 4 байта).

4. Атрибут *позфнс* определяет *позицию номера* в строке: 0 – *в начале* строки (диск, барабан), (*длинстрок–длинфнс*) – *в конце* строки (перфокарты).

5. Атрибут *пакстрок* учитывает возможность *упаковки пробелов*. Как известно, в системе ОС ЕС ЭВМ одна строка текстового файла занимает 80 байтов независимо от числа символов, отличных от пробела. Эта особенность приводит к неэкономному расходу внешней памяти, частым переполнениям библиотек и даже к ухудшению стиля программирования из-за стремления программистов вместить в строку как можно больший объем информации. В системе "Эльбрус" предусмотрена упаковка подряд идущих пробелов в текстовом файле: *k* пробелов заменяются одним байтом со значением *k* (*k* < = 63). Если *пакстрок* = 1 (это значение принято по умолчанию), то в файле могут встретиться упакованные пробелы. Все систем-

ные программы (редактор, трансляторы и т.п.) могут работать с упакованными текстовыми файлами. При *пакстрок* = 0 система "понимает" обычный неупакованный файл (например, полученный после переноса с другой ЭВМ), который после модификации атрибута *пакстрок* при первой же редакции автоматически упаковывается. По экспериментальным данным, размер файла при упаковке пробелов уменьшается на 10–20%.

С т р у к т у р а т е к с т о в о г о ф а й л а. На барабане и диске стандартной является следующая структура. Файл состоит из строк переменной длины (*длинстрок* = 0). Строки упаковываются в блоки файла; строка не может переходить из одного блока в другой. Структура строки:

дл ном строка,

где *дл* – байт длины, *ном* – номер строки, упакованный в двоичном виде в четыре байта (*длинфнс* = 8, *кодфнс* = 0), *строка* – символы строки. Значение *дл* учитывает общую длину строки, включая байты *дл* и *ном* (следовательно, *дл* > 0). Значение *дл* = 0 – признак конца блока, если за последней строкой его есть свободные байты.

Система атрибутов текстового файла позволяет создать и файлы другой структуры (например, ненумерованные со строками одинаковой длины).

1.27.2. Изображение текстового файла. Для текстовых файлов в языке Эль-76 имеется возможность *изображения*. Например,

```
начало
    конст данные = тфайл * конецданных *
                  1   –5  0  62  13  7
                –98   21  4  35   2  11
          конецданных; . . .
конец
```

В этой программе содержится изображение *вложенного текстового файла* из двух строк. Этот файл наследует все атрибуты файла текста программы. Изображение начинается словом **тфайл**, за которым следует *определение конечного ограничителя* в форме: *ses* (*s* – разделитель, *e* – последовательность символов); в примере – * *конецданных* *. Последовательность символов *конецданных* воспринимается транслятором как конец вложенного текстового файла. Поэтому не рекомендуются короткие последовательности ("!", "*все*" и т.п.), которые могут случайно встретиться в тексте. Идентификатор *данные* ссылается на изображенный текстовый файл и может использоваться для операций над ним, например: *позп* (*данные*).

Изображение текстового файла используется главным образом при составлении *пакетных заданий* (п. 1.30.3) для обозначения текстов, исходных данных, директив редактора и т.п.

1.27.3. Пример работы с текстовыми файлами. Для иллюстрации стиля обработки текстовых файлов, принятого в СПФ, рассмотрим решение следующей задачи: напечатать содержимое строки заданного файла с заданным номером, а если ее нет – диагностическое сообщение. Предполагается, что текстовый файл расположен на диске или барабане и имеет стандартную структуру (п. 1.27.1).

```
процедура печатьстроки = проц (ф, н)
 (ф64печ:= читатрзадачи (виртвыв),    % файл печати
       п:=лозп (ф, формэлем: 3),       % позп на файл ф
       блок:=чит (ф,0),                % указатель блока
       строка,                         % указатель строки
       номер,                          % номер строки
       длина,                          % длина строки
       счет,                           % счетчик пробелов
       буф:=лок вект [128] ф8,         % буфер для печати
       текбуф:= буф;                   % текущий указатель
 % --- поиск строки с заданным номером --- %
 до    *ситлкф, найден цикл
       % -- проверка на конец блока; переход к следующему
       если если длина блок = 0 то истина
            иначе блок [0] = 0
            все
       то блок:=чит (п, след)
       все;
       % -- выделение очередной строки и ее номера -- %
       строка:= блок [: (длина:= блок [0])];
       блок:= блок [длина:];
       номер:=строка [1:4]@;
       если номер = н то найден! все
повторить
при   найден: % --- печать найденной строки --- %
      строка:= строка [5:]; % отсечение длины и номера
      счет:= 0;
      до   конецпечати цикл
           если счет = 0 то % -- нет упакованных пробелов
              мод текбуф <:= мод строка пока >="";
              если тгп % исчерпан буфер печати
              то запф (печ, буфер, :l); текбуф:= буфер
              инес тги % исчерпана строка
              то запф (печ, буфер [: 128-длина текбуф],:l);
                 конецпечати!
              иначе % упакованные пробелы; распаковка и печать
                    счет:= строка [0];
                    строка:= строка [1:]
              все
           иначе % печать упакованных пробелов
             мод текбуф <:= заполн "" длиной мод счет;
             если тгп то запф (печ, буфер,: l); текбуф:= буфер
             все
           все
      повторить,
ситлкф: % --- конец файла: не найдена строка с номером
        запф (печ, стр8 "не найдена строка номер", н: g (-8))
всесит;
открeп (п)) % --- печатьстроки --- %
```

1.28. Ошибочные ситуации при обмене

В заключении описания системы простых файлов приведем перечень основных ситуаций, возникающих при обмене. Любая из этих ситуаций может быть обработана структурным предложением. В необходимых случаях указаны возможная причина ошибки и рекомендации по ее исправлению.

1. *ситлкф* – логический конец файла при буферизованном обмене; попытка чтения или установки блока файла, который еще не заполнен информацией.

2. *ситфкф* – физический конец файла; превышение значения атрибута *максдлинфайла* при записи в файл. Рекомендуется увеличить значение атрибута и повторить обмен (см. п. 1.21.3).

3. *ситнетлиста* – попытка чтения из еще не созданного листа файла на барабане или на диске. Возможная причина: неверно задан номер блока или сектора в операции чтения. Такая ошибка часто возникает при запуске неверно сформулированного файла объектного кода.

4. *ситпривобмен* – попытка обмена значениями класса адресной информации. Рекомендуется изменить представление записываемых в файл данных, исключив из них адресную информацию (например, введя относительные адреса в виде индексов).

5. *ситошпарамобмена* – неверное обращение к операции обмена: неверный порядок параметров, попытка работы с последовательным устройством как с устройством произвольного доступа и т.п. Например, эта ситуация возникает в операции: *чит* (*экран,* 10).

6. *ситнетвнеш* – нет внешнего объекта: попытка обращения к внешнему объекту после его уничтожения. Например:

ликвиднеш (*ф*); *n*:= *позп* (*ф*)

7. *ситнетрес* – нет ресурса: отсутствие свободного места в контейнере для создания файла. Если эта ситуация возникает при редактировании текста, то исчерпано пространство на барабане; рекомендуется выйти из диалога и вновь войти в него. Если же ситуация возникла при создании файла на диске, то для размещения нового файла может потребоваться удаление некоторых других файлов.

8. *ситошатр* – ошибка в атрибуте: обращение к несуществующему атрибуту внешнего объекта. Одна из типичных ошибок – попытка чтения атрибута *имяфайла* у дискового или барабанного файла.

9. *ситошву* – ошибка внешнего устройства: устойчивые сбои при многократных попытках обмена. Рекомендуется проверить устройство и повторить обмен.

10. *ситплохойконт* – нарушение структуры контейнера на диске или на ленте. Рекомендуется переставить носитель на другое устройство, а при устойчивом эффекте обратиться к системному программисту для ремонта контейнера.

11. *ситошпарамарх* – ошибка в параметрах архивной операции, например: *элспр* (*генув* ( ), "ф", *спр*)

12. *ситнетварх* – отсутствие заданного элемента в справочнике. На практике эта ситуация часто возникает при неверном задании внешнего имени,

например, при пропуске слога. Рекомендуется проверить все внешние имена в тексте программы.

13. *ситнетархслк* – нет архивной ссылки: пустая ссылка *пустовнеш* в элементе справочника. Эта ситуация возникает сравнительно редко, так как элемент справочника, как правило, создается с определенным значением (ссылкой на внешний объект).

1.29. Некоторые особенности семантики СПФ

В этом параграфе рассмотрены особенности системы простых файлов, которые, по опыту ее практического использования, оказались наиболее непривычными. При использовании СПФ программист сталкивается с ними редко, однако в системных программах их требуется учитывать.

1.29.1. Идентификация внешних объектов. При выполнении операций обмена система файлов контролирует типы внешних объектов. Однако в программе такой контроль может потребоваться и в явном виде.

Внешний объект всегда доступен через оперативный объект связи – подвижный указатель, внешнее имя, позиционную переменную и т.п. Если *x* – ссылка на внешний объект, то значением атрибута *типоб* (тип объекта) является тип оперативного, а не внешнего объекта *x*. Например, если

читатр (*x*, *типоб*) = 2

то *x* – заголовок открытого файла, при значении 3 – позиционная переменная, при значении 8 – подвижный указатель, 9 – внешнее имя. Для значений *x*, равных *ксн*//*лгу* (внешнее имя контейнера) и *ксн*//*лгу*//*текст* (внешнее имя файла), значение атрибута *типоб* одно и то же и равно 9.

Тип внешнего объекта, на который ссылается оперативный объект *x*, можно получить с помощью атрибута *типвнешоб*. Для файла его значение равно 2, для справочника – 10. Например:

```
читатр (ксн//лгу, типвнешоб) = 10 % указывается базовый справочник
                                    контейнера
читатр (ксн//лгу//текст, типвнешоб) = 2
```

Другой возможный способ идентификации внешнего объекта – попытка выполнения характерных для него операций с обработкой возможных ошибочных ситуаций:

```
текст этоконтейнер (x) = % -- x – контейнер? -- %
      до * ситошпарамобмена
          (ф64 к := контейнер (x);
          откреп (к);
          истина )
          при ситошпарамобмена : ложь
          всесит
```

Другая типичная задача такого рода – сравнение ссылок на внешние объекты – решается операцией *тотжефобъект*: выражение

тотжефобъект (*x*, *y*)

истинно в том и только в том случае, если x и y ссылаются на один и тот же внешний объект.

1.29.2. Особенности обращения к атрибутам. В СПФ различаются следующие классы атрибутов:

1. *атрибуты внешнего объекта*, хранящиеся в его заголовке; например, атрибуты файла *типву* и *типфайла*;

2. *атрибуты открытия*, хранящиеся в оперативном объекте связи, например, атрибут *формэлем* позиционной переменной.

По заданному оперативному объекту связи можно получить или модифицировать значение атрибута внешнего объекта. Например, если

ф := *генфайл* (*барабан*);
п := *позп* (*ф*),

то *читатр* (*п*, *типву*) = 1 (атрибут *типву* берется из файла *ф*). Операция *запатр* (*п*, *типфайла*: 0) изменяет значение атрибута файла *ф* на число 0. Ситуация *ситошатр* (ошибка атрибута) возникает, лишь если ни у оперативного объекта, ни у связанного с ним внешнего объекта такого атрибута нет.

Атрибут открытия можно задавать как в генераторе внешнего объекта, так и собственно при открытии:

ф1 := *генфайл* (*ксн*//*лгу*, *формэлем*: 3); % в генераторе
п1 := *позп* (*ф1*, *формэлем*: 0); % при открытии

Если при открытии атрибут не задан, берется его умолчание из заголовка файла *ф1*, равное 3, а если оно не указано при генерации, то стандартное значение по умолчанию, равное 6.

1.29.3. Связь различных форм ссылок на объекты. В СПФ существуют две формы ссылок на внешний объект:

1. *подвижный указатель*, устанавливаемый на объект или элемент справочника:

ф := *генфайл* (*ксн*//*лгу*);
у := *элспр* (*генув* ( ), *с*, "ф1")

При создании элемента справочника *у* подвижный указатель создается явно, операцией *генув* (указатель, созданный операцией *генув*, устанавливается на заданный элемент справочника) при создании файла *ф* – неявно (этот указатель и является результатом операции *генфайл*);

2. *план поиска* (на языке Эль-76 изображаемый конструкцией *внешнее имя*), например:

в := *ксн*//*лгу*//*иванов*//*текст*1

В основных операциях над объектами (открытие, закрытие, обращение к атрибутам, операции обмена) эти две формы ссылок совершенно равноправны. Однако следует иметь в виду, что наиболее "глобальной" из них является внешнее имя, через которое, в конечном счете, обеспечивается сохранение внешних объектов. Например, если ссылку *ф* не занести ни в какой справочник, то при завершении задачи этот файл будет уничтожен.

Запись ссылки на внешний объект в архив осуществляется операцией:

создэлспр (*с*, "ф", *ф*)

или:

запспр (*элспр* (*генув* ( ), *с*, "ф1"), *ф*)

Для "разыменования" внешних имен с помощью подвижных указателей служит операция:

обкт (*ув*, *в*)

где *ув* – подвижный указатель, *в* – любая ссылка на внешний объект или на элемент справочника (например, внешнее имя). Операция *обкт* устанавливает указатель *ув* на тот внешний объект, на который ссылается *в*. При этом, если необходимо, осуществляется поиск по внешнему имени. Если же *у* – ссылка на элемент справочника (см. выше), то операция *обкт* (*ув*, *у*) устанавливает указатель *ув* на тот же объект (файл), на который ссылается элемент справочника *у*.

Резюмируя, можно сказать, что нетрадиционной особенностью СПФ является наличие нескольких форм ссылок на внешние объекты. Такого рода ссылки неявно присутствуют в реализации любой файловой системы, но средства управления ими обычно программисту не предоставляются. Включение этих средств в язык Эль-76 вполне соответствует принципу универсальности базового языка в системе "Эльбрус".

1.30. Взаимодействие с операционной системой

В этом параграфе рассмотрим способы управления единицами вычислений в языке Эль-76. В операционной системе ОС "Эльбрус" различаются следующие единицы вычислений:

1. *программа* – независимо транслируемая единица вычислений, текст которой независимо транслируется в отдельный *файл объектного кода* (п. 1.30.1);

2. *процесс* – единица вычислений, имеющая свое независимое *управление*, которая может исполняться параллельно другим процессам (п. 1.30.2);

3. *задача* – наиболее крупная единица вычислений, в рамках которой производится распределение *ресурсов*, прежде всего – *математической памяти* (п. 1.30.4). Задача может исполняться в диалоговом или пакетном режиме. В задаче может быть запущено несколько процессов.

Кроме того, традиционно вводится также понятие *задание* – текст программы пакетной задачи.

Управление всеми этими единицами вычислений в системе "Эльбрус" не требует какого-либо особого языка, а осуществляется на базовом языке Эль-76.

1.30.1. Управление программами. Программа имеет независимый *файл объектного кода*, который может быть получен либо компилятором, либо в результате *комплексации* (сборки) из процедур, написанных на одном или нескольких языках. Основные операции над программой в системе

"Эльбрус" – *открытие* и *выполнение.* Пусть *код* – ссылка на файл объектного кода программы. Операция

прогр (*код*)

осуществляет *открытие программы* – подготовку её к исполнению. Результатом операции *прогр*, как правило, является процедурная *метка*, по которой программа может быть запущена как обычная процедура. Например, если текст программы имеет вид определения процедуры:

пример = проц (*x*) начало . . . конец

а файл объектного кода программы имеет внешнее имя *ксн//паскаль//к,* то программа *пример* может быть запущена следующим образом:

p := *прогр* (*ксн//паскаль//к*) ; % открытие
p (1)   % запуск

либо

прогр (*ксн//паскаль//к*) (1)   % открытие и запуск

Программа – закрытое предложение запускается как процедура без параметров: *p* ( ) , где *p* – метка программы.

Программа-пакет (п. 1.18.3) отличается тем, что результатом ее открытия является константный справочник. К его элементам – меткам процедур – можно обращаться с помощью упрощенной формы операции *элспр* (п. 1.23.1) :

пакет := *прогр* (*ксн//оспок3//кодфун64*) ;
% стандартный пакет элементарных функций
sin := элспр (пакет, "sin") ;
  . . . sin (*x*) . . .

Открытие программы отличается от традиционной загрузки тем, что объектный код программы не считывается из ФОК в оперативную память. При открытии создается лишь специальный вектор – *словарь сегментов кода*, содержащий ссылки на сегменты объектного кода программы в ФОК. Каждый сегмент, как правило, соответствует процедуре. При первом обращении к сегменту он *динамически* загружается в оперативную память (аналогично вектору). Справочник сегментов кода программы хранится в единственном экземпляре, даже если программа открывается несколько раз из разных задач. Таким образом, в памяти (в одном экземпляре) хранятся только те части программы, к которым происходит обращение. Такой способ подключения независимой программы называется *динамическим знакомством* и придает языку Эль-76 важное для языка системного программирования свойство *расширяемости.*

При динамическом знакомстве программа, в которой выполняется операция *прогр* (*p*), как бы динамически расширяется программой *p.* Тем самым, к любой программе могут по мере необходимости динамически подключаться другие независимо транслируемые программы. Например, многопросмотровый компилятор может состоять из независимых программ-просмотров, обменивающихся информацией через парамет-

ры и результаты, и головной программы сборки:

```
компилятор = проц (текст, код)
начало
   ф64 просм1 := прогр (//кодпросм1), % просмотр 1
       просм2 := прогр (//кодпросм2), % просмотр 2
       просм3 := прогр (//кодпросм3), % просмотр 3
       внутр1, внутр2; % внутреннее представление
   внутр1 := просм1 (текст);
   внутр2 := просм2 (внутр1);
            просм3 (внутр2, код)
конец % --- компилятор --- %
```

1.30.2. Управление процессами. Система "Эльбрус" – многопроцессорный вычислительный комплекс, в котором имеется реальная возможность распараллеливания программ (конфигурация системы может включать до 10 центральных процессоров). В операционной системе реализованы три класса процессов:

1. *подчиненный процесс* – процесс, запускаемый параллельно с исполнением создающей его программы и *локальный* в некотором *критическом блоке* (выход из этого блока осуществляется только после окончания подчиненного процесса);

2. *независимый процесс* – процесс, запускаемый параллельно с создающей его программой и не связанный ни с каким критическим блоком (*глобальный*);

3. *сопроцесс* – процесс, создаваемый с пассивном состоянии и запускаемый с помощью системной процедуры *возобновить.* Этот класс процессов предназначен для организации сопрограммного режима работы в стиле языка Симула-67. Сопроцесс, как и подчиненный процесс, является локальным.

Перед созданием подчиненных процессов или сопроцессов в области локальных данных критического блока необходимо записать *документацию на локальные процессы.* Для этого следует описать переменную формата **ф64**:

ф64 *док*

и вызвать системную процедуру формирования документации:

документпроцесс (**имя** *док*)

Переменную *док* (имя которой указывается при создании процесса) ОС использует для хранения счетчика подчиненных процессов и сопроцессов, созданных в критическом блоке.

С о з д а н и е   п а р а л л е л ь н о г о   п р о ц е с с а осуществляется вызовом системной процедуры:

создпроцесс (*приор, длстека, класс, адрдок, p, x1, . . . , xk*)

Смысл параметров:

1. *приор* – приоритет процесса. С каждым процессом в системе связывается *приоритет* (целое положительное число). Максимальный возможный приоритет равен 100; приоритет порожденного процесса не может быть

больше, чем у процесса-родителя (иначе возможна блокировка). При наличии свободного процессора первым будет исполняться процесс с наивысшим приоритетом.

2. *длстека* – длина стека процесса. Каждому процессу система выделяет *стек* для исполнения его процедур. Для стека отводится область физической памяти длиной min (*длстека, m*) слов, где *m* – внутренняя константа ОС.

3. *класс* – класс процесса: 1 – *подчиненный*, 2 – *независимый*, 3 – *сопроцесс*.

4. *адрдок* – для подчиненных процессов и сопроцессов – адрес документации на процессы (в форме: **имя** *док*), для независимых процессов – 0.

5. *p* – метка головной процедуры, используемой в качестве процесса; *x1, ..., xk* – ее фактические параметры.

Результат процедуры *создпроцесс* – указатель процесса, используемый для управления им. Если создается подчиненный или независимый процесс, то он ставится в очередь к процессору и при наличии свободного процессора запускается (с учетом приоритета).

Управление сопроцессом осуществляется с помощью *указателя процесса*:

x := *создпроцесс* (. . .)

Запуск сопроцесса выполняется системной процедурой:

возобновить (*x, k*)

которая запускает процесс *x* и приостанавливает исходный процесс. Параметр *k* равен 0 или 1; 1 – исходный процесс не будет больше работать, 0 – возможно возобновление приостановленного текущего процесса. *Указатель на текущий процесс* выдается системной процедурой *мойстек* ( ).

Процедуры управления процессами. Если *x* – указатель процесса, то:

1. процедура *приостановитьпроцесс* (*x*) переводит процесс *x* в состояние "приостановлен";

2. процедура *активизироватьпроцесс* (*x*) выполняет обратную операцию активизации приостановленного процесса *x*;

3. процедура *убитьпроцесс* (*x*) уничтожает процесс *x*.

Примеры.

1. Организация подчиненного процесса.

```
начало % критического блока
   ф64 докум,          % документация на процессы
       укпроцесс;      % указатель процесса
   процедура процесс = проц ( ) ( . . . );
   % создание документации на процесс %
   докумнапроцесс (имя докум);
   % запуск процесса %
   укпроцесс := создпроцесс (1, 512, 1, имя докум, процесс)
   % уничтожение процесса выполняется неявно
конец
```

```
начало % критического блока
   ф64 доким; % документация на процессы
   ф64 процесс1, процесс2; % указатели на сопроцессы
   процедура проц1 = проц (осн)
   % тело сопроцесса 1. осн – указатель процесса основной
   % программы.
      начало % сопроцесса 1
         ... % запуск сопроцесса 2: %
         возобновить (процесс2, 0) ...
      конец; % сопроцесса 1
   процедура проц2 = проц (осн) % тело сопроцесса 2
      начало % сопроцесса 2
         ... % запуск сопроцесса 1 : %
         возобновить (процесс1, 0) ...
      конец; % сопроцесса 2
% тело основной программы. создание документации на процессы:
докумнапроцесс (имя докум);
% создание сопроцессов 1 и 2 : %
процесс1 := создпроцесс (1, 512, 3, имя докум, проц1, мойстек ( ));
процесс2 := создпроцесс (1, 512, 3, имя докум, проц2, мойстек ( ));
% запуск сопроцесса 1 : %
возобновить (процесс1, 0) ...
конец % критического блока
```

Выход из сопроцессов 1 или 2 в основную программу может быть осуществлен вызовом системной процедуры *возобновить* (*осн*, 1). По завершении основной программы созданные в ней сопроцессы 1 и 2 автоматически уничтожаются.

Синхронизация процессов. В языках программирования известно большое число методов синхронизации процессов – семафоры, мониторы, рандеву, порты и т.п. В системе "Эльбрус" аппаратно реализовано наиболее простое базовое средство синхронизации – *двоичные семафоры. Семафор* – системный объект, который может находиться в двух состояниях – "открыт" и "закрыт", причем операция открытия или закрытия выполняется как неделимый акт, во время которого обеспечивается монопольный доступ к семафору.

Рассмотрим реализацию с помощью семафоров на языке Эль-76 двух основных схем синхронизации – по ресурсам и по событиям.

Синхронизация по ресурсам. Пусть имеется классическая задача взаимного исключения: совокупность процессов обращается к общему ресурсу; требуется обеспечить, чтобы в каждый момент доступ к ресурсу имел только один процесс.

С ресурсом связывается первоначально открытый семафор *c*, который создается как копия значения стандартной константы *открытый*:

```
ф64 c := открытый
```

В каждом процессе выделяется *критический интервал*, в котором осуществляется доступ к ресурсу, и заключается в *семафорные скобки:*

закрыть (**имя** *с*);
% критический интервал: обращение к ресурсу
открыть (**имя** *с*);

Операция *закрыть*, выполненная каким-либо процессом над открытым семафором *с*, закрывает его, и процесс получает доступ к ресурсу. Другие процессы, пытаясь закрыть уже закрытый семафор, приостанавливаются в ожидании его открытия. Операция *открыть*, выполненная по завершении критического интервала, возобновляет все процессы, приостановленные на семафоре *с*, после чего вновь один из них закрывает его, и т.п. Отметим, что наиболее часто встречающиеся варианты операций (закрытие открытого семафора, открытие семафора без приостановленных процессов) выполняются аппаратными операциями без обращения к ОС. Описанная схема синхронизации используется в ОС "Эльбрус" для выделения оперативной памяти для параллельно исполняемых задач.

При синхронизации процессов следует учитывать, что семафоры сами по себе не предохраняют от блокировки процессов. Например:

процесс1 = **проц** ( )
    начало
        закрыть (**имя** *с1*); *закрыть* (**имя** *с2*); *P*;
        открыть (**имя** *с1*); *открыть* (**имя** *с2*)
    конец
процесс2 = **проц** ( )
    начало
        закрыть (**имя** *с2*); *закрыть* (**имя** *с1*); *Q*;
        открыть (**имя** *с2*); *открыть* (**имя** *с1*)
    конец

Здесь процессы *процесс1* и *процесс2* при параллельном исполнении могут заблокировать друг друга (если *процесс2* успел закрыть семафор *с2* раньше, чем *процесс1* закрыл этот семафор, но после того, как этот процесс закрыл семафор *с1*), так как первый процесс бесконечно ждет открытия семафора *с2*, второй – семафора *с1*, и критические интервалы *P* и *Q* выполнены не будут.

Синхронизация по событиям. Другая типичная задача синхронизации формулируется следующим образом. Требуется, чтобы каждый из параллельных процессов в некоторый момент ожидал наступления события *с*, а после его наступления все процессы продолжали работу. Этот тип синхронизации реализуется двумя другими операциями над семафором: *ждать* и *пропустить*.

Событие *с* будем представлять закрытым семафором:

ф64 *с* := *закрытый*;

Точка ожидания события *с* в каждом процессе помечается вызовом операции:

ждать (**имя** *с*)

Операция *ждать* приостанавливает процесс на закрытом семафоре *с* (что соответствует необходимости ожидания события). Если же семафор *с* открыт, то процесс продолжается (это означает, что ожидания события не требуется).

Процесс, ответственный за наступление события, сигнализирует о нем операцией *пропустить*:

пропустить (**имя** *с*)

Таким образом, выполнение операции *пропустить* (**имя** *с*) вызывает завершение операции *ждать* (**имя** *с*) во всех приостановленных на семафоре *с* процессах, но семафор *с* остается закрытым.

1.30.3. Управление заданиями. Как уже отмечалось, функция языка управления заданиями в системе "Эльбрус" выполняет язык Эль-76. В нем есть все элементы, необходимые для программирования заданий:

– операции открытия и запуска программы (п. 1.30.1);

– средства создания и именования внешних объектов (пп. 1.21, 1.23);

– средства изображения текстовых файлов (исходных данных, тестов, директив редактора, п. 1.27.2);

– средства вызова системных программ (как стандартных процедур с предопределенными идентификаторами):

1) текстовый редактор:

редактор (*фоткуда*, *фред*, *фкуда*)

Здесь: *редактор* – идентификатор программы; *фоткуда* – исходный файл для редакции; *фред* – файл директив редактора (например, терминальный файл); *фкуда* – файл для записи результата редакции;

2) транслятор Эль-76:

автокод (*текст*, 0, *код*)

Здесь: *автокод* – идентификатор программы (от первого названия языка – Автокод Эльбрус [52]); *текст* – файл исходного текста программы; *код* – файл для записи ФОК. Результат процедуры-функции *автокод* – число ошибок в программе.

Программа задания после ввода транслируется компилятором Эль-76 и в случае отсутствия ошибок исполняется.

П р и м е р з а д а н и я. Пусть в текстовом файле //*текст* содержится текст программы. Над ним требуется выполнить следующую редакцию:

– в строке 00100 заменить символ " , " на символ " ; ";

– вставить строку 00311, содержащую слово **все**.

Результат редакции требуется записать в исходный файл. Полученный текст транслировать и при отсутствии ошибок запустить код программы // *код* на исполнение без параметров. Файлы исходного текста и кода программы имеют имена "текст" и "код" в справочнике *ксн*//*паскаль*//*тесты*.

Программа задания на языке Эль-76, выполняющая указанные действия, имеет следующий вид:

```
начало
   конст c = ксн//паскаль//тесты, % справочник
         дир =  тфайл * кондир *
                з 00100 / , / ; /
                00311 – все
                кондир; % файл директив
   редактор (c // текст, дир, 0);
   если автокод (c // текст, 0, c // код) = 0
   то прогр (c // код) ( )
   все
конец % ––– задания ––– %
```

1.30.4. Управление задачами. Новая задача в системе "Эльбрус" создается следующими способами:

1) при вводе пакетного задания, в случае его успешной трансляции (*пакетная задача*);

2) при входе пользователя в диалог под зарегистрированным в системе именем с пользовательского терминала (*диалоговая задача*);

3) в результате исполнения конструкции "*запуск задачи*".

Для созданной задачи система выполняет независимое распределение математической памяти.

Таким образом, математический адрес из одной задачи нельзя использовать в другой задаче. В пределах одной задачи могут создаваться процессы и запускаться программы, использующие общее адресное пространство.

Атрибуты задачи. Задача во время исполнения характеризуется набором *атрибутов*. Значение атрибута текущей задачи может быть получено конструкцией

читатрзадачи (*имя*) ,

где *имя* – имя атрибута задачи. Наиболее часто используемые атрибуты задачи – *виртвыв* (виртуальный вывод) и *времяцп* (текущее значение времени исполнения задачи на центральном процессоре).

Значение атрибута *виртвыв* – позиционная переменная на *файл виртуального вывода задачи*. Этот файл хранится на барабане, а при завершении пакетной задачи или при выполнении директивы печати в диалоговой задаче выводится на АЦПУ. Все системные программы используют файл виртуального вывода для вывода протокола своей работы. Он же служит для вывода на АЦПУ в пользовательских программах. Примеры его использования приводятся на протяжении всей главы.

Атрибут *времяцп* предназначен для измерения времени исполнения программы или отдельных ее фрагментов. Поскольку значение этого атрибута выдается в единицах времени центрального процессора, для удобства введена стандартная константа *секунда* – число единиц времени центрального процессора в одной секунде.

П р и м е р.

```
% начало отсчета времени: %
время := читатрзадачи (времяцп);
% исполнение фрагмента программы %
% конец отсчета времени: %
время := (читатрзадачи (времяцп) – время) / секунда;
запф (печ, стр 8 "время счета = ", время:g (–6,2))
```

Атрибуты задачи *максматпам, максвремяцп, максвремяобмена, максразмацпу* предназначены для установки лимита на соответствующие ресурсы, что может оказаться полезным для контроля за их лавинообразным использованием и зацикливанием задачи. По умолчанию никаких существенных ограничений на задачу не накладывается (например, максимальный объем математической памяти задачи равен 2^{32} слов).

Атрибуты *имязадачи* и *имяпол* задают имя задачи и имя пользователя.

З а п у с к   з а д а ч и. Создание новой задачи и ее запуск осуществляются конструкцией "запуск задачи". Например:

```
задача (имязадачи: "печ", максвремяцп: 100 * секунда,
        максразмацпу: 25)
печать = проц (файл)
% печать чисел из файла до его исчерпания или до
% превышения указанных лимитов
  начало
    конст печ = читатрзадачи (виртвыв),
          ввод = позп (файл);
    до *  ситлкф цикл
          ф64 x;
          читф (ввод, имя x);
          запф (печ, x, : l)
    повторить
  конец (ксн//паскаль//числа)
```

За словом **задача** следует *список установки атрибутов*, затем – *текст программы* и *список* ее *фактических параметров*. Для запуска задачи система создает независимый процесс, в котором запускается заданная головная процедура задачи с указанным набором параметров. Конструкция "запуск задачи" используется, главным образом, операционной системой.

1.31. Средства модульного программирования

Ядро языка Эль-76 содержит процедуры (п. 1.16) и текстовые макросы (п. 1.17), которые можно считать простейшими средствами модульного программирования. Однако это понятие, в его современном смысле, значительно шире. Важнейшими чертами модульности являются:

1. *Пакеты* – описания совокупности данных и процедур, состоящие из видимой (декларативной) и скрытой (реализационной) частей. Понятие пакета (package) введено в язык Ада и, в несколько иной форме, в язык

Модула-2. Компоненты видимой части пакета *p* доступны через составные имена: *p. x*.

2. *Контекст компиляции* – указание контекста идентификаторов, которые могут использоваться в некотором модуле. Эти идентификаторы берутся из интерфейса одного или нескольких пакетов. В языке Ада контекст компиляции задается предложением использования (**use**) в языке Модула-2 – списком импорта (**from** ... **import**).

Средства модульного программирования, являющиеся развитием этих концепций в духе общих принципов динамизма системы "Эльбрус", введены в качестве расширения в язык Эль-76.

1.31.1. Модульный объект, его интерфейс и реализация. Основная конструкция модульного программирования в языке Эль-76 – *модульный объект*. Понятие модульного объекта близко концепции пакета языка Ада, но обеспечивает бóльшую гибкость при программировании. Для краткости в этом параграфе вместо термина "модульный объект" будем иногда использовать термин "модуль".

Описание модуля. Модуль состоит из *интерфейса* и *реализации*, которые являются единицами компиляции наряду с закрытым предложением и определением процедуры. Например:

```
интерфейс     % —— модульного объекта "стек" —— %
ф64           % пример рекомендуемого комментария к описанию
                интерфейса
взять,        % ( ) –> вершина стека
              % вычеркивает вершину стека и выдает ее содержимое
              % в качестве результата. Возможна ситуация исчерпан,
              % если стек пуст.
верх,         % ( ) –> вершина стека
              % выдает содержимое вершины стека. Возможна ситуация
              % исчерпан, если стек пуст.
положить,     % (x)
              % помещает объект x в вершину стека. Возможна
              % ситуация переполнен (переполнение стека)
пуст,         % ( ) –> признак "стек пуст"
              % результат – истина, если стек пуст, иначе ложь
исчерпан,     % ситуация "исчерпание стека" (возникает при попытке
              % обращения к вершине пустого стека)
переполнен    % ситуация "переполнение стека" (возникает при
              % превышении размера стека)
конинт % —— внешнее имя кода интерфейса – // стек —— %
реал // стек
  начало
  % —— описание скрытой части реализации —— %
    конст длстека = 1000;
    ф64 с := лок вект [длстека] ф64,
        ус := 0;
    процедура
    рвзять= функция ( ) % реализация операции "взять"
```

```
        если ус = 0 то исчерпан!
        иначе с [(ус := *– 1) + 1]
        все, % –– рвзять –– %
    рпуст = функция ( ) % реализация операции "пуст"
        (ус = 0), % –– рпуст –– %
    рположить = проц (х) % реализация операции положить
    если ус = длстека то переполнен!
    иначе с [ус := *+ 1] := х
    все,% –– рположить –– %
рверх = функция ( ) % реализация операции верх %
    если ус = 0 то исчерпан! иначе с [ус] все; % – рверх – %
% инициализация интерфейса %
взять := рвзять; верх := рверх; положить := рположить;
пуст := рпуст; исчерпан := ситуация ( );
переполнен := ситуация ( )
конец % ––– реализации модуля "стек". Внешнее имя кода: // рстек.
```

Интерфейс модуля состоит из последовательности описаний переменных и констант. Объекты, входящие в интерфейс, доступны пользователю модуля. В интерфейс модуля *стек* входят шесть переменных; четыре из них используются для ссылок на операции (процедуры), две – на исключительные ситуации. Интерфейс может быть транслирован в отдельный файл объектного кода, внешнее имя которого используется для ссылки на интерфейс в тексте программы.

Для заданного интерфейса может быть описана одна или несколько реализаций. В примере приведена реализация стека в виде массива. *Изображение реализации* содержит *имя интерфейса* и *закрытое предложение,* в котором описываются элементы *скрытой части* реализации – конкретное представление объекта и реализации (тела) операций. Изображение реализации завершается последовательностью операторов, в которой, как правило, выполняется *инициализация интерфейса.* Изображение реализации, как и интерфейс, может быть транслировано в отдельной ФОК (в примере – *// рстек*).

Отделение интерфейса от реализации обеспечивает возможность пошаговой разработки программ. Интерфейс определяется и компилируется на этапе проектирования программы. Написание и компиляция других модулей, использующих модуль *// стек,* может выполняться до его реализации или параллельно с ней. Модуль с заданным интерфейсом может иметь несколько реализаций; например, модуль *// стек* может быть реализован с использованием конкретного представления стека в виде связанного списка.

Использование модуля. В языке Эль-76 модуль рассматривается как объект. Типом этого объекта считается описание интерфейса модуля. По заданным описаниям интерфейса и реализации можно с помощью *генератора модульного объекта* создать произвольное число его *экземпляров* заданной структуры:

конст *с* # *// стек* = *(// рстек)*

Форма описания константы, применяемая для модульных объектов, в левой части содержит идентификатор константы и имя интерфейса

модуля, которое определяет способ доступа к компонентам объекта, их имена и порядок. Правая часть описания — *генератор модуля*, параметром которого является внешнее имя кода реализации. Генератор создает экземпляр интерфейса и связанный с ним экземпляр реализации. В терминах ядра Эль-76 это означает, что создаются связанные друг с другом области памяти для размещения интерфейса и локальных данных реализации. В примере значениями переменных интерфейса будут метки процедур и указатели ситуаций, область реализации инициализируется в соответствии с ее описанием. Процедуры, входящие в интерфейс (например, процедура *взять*), определены и исполняются в контексте созданных экземпляров интерфейса и реализации, т.е. в процедуре *взять* используются те самые экземпляры переменных *исчерпан*, *ус* и *с*, которые созданы генератором.

С точки зрения программиста, создан новый экземпляр объекта *//стек*, который получил имя *с* и доступен через интерфейс. Обращения к компонентам интерфейса стека *с* выполняются через *составные имена*: *с.взять*, *с.положить* и т.п. Например:

с.положить (1); *с.положить* (2); *с.положить* (3);
запф (*печ*, *с.взять* ( ), *с.взять* ( ), *с.взять* ( ), : *l*)

В результате будет напечатано: 3 2 1.

1.31.2. Подключение контекста модуля. Форма использования модуля *// стек*, описанная в п. 1.31.1, удобна в случае, если в программе используются одновременно несколько стеков:

конст *с1* # *// стек* = (*// рстек*),
с2 # *// стек* = (*// рстек*); ...
с1. положить (1); *с2.положить* (0)

Если же используется лишь один стек *с*, постоянное повторение префикса "*с*." неудобно. В этом случае применяется другая форма описания стека:

контекст конст *с* # *// стек* = (*// рстек*); *с*

В результате этого описания устанавливается контекст идентификаторов интерфейса, и префикс "*с*." при обращении к его компонентам можно опускать:

положить (1); *запф* (*печ*, *взять* ( )).

В одном блоке возможно несколько подключений контекста различных модулей. Идентификатор *с* служит для использования составных имен в случае конфликтов.

1.31.3. Параметры интерфейса и реализации. Как интерфейс, так и реализация могут иметь *параметры*, которые используются для указания начальных значений величин. Например, интерфейс модуля *// стек* может быть описан с параметром, задающим указатель ситуации *исчерпан*:

интерфейс (*с*)
ф64 *взять*, *верх*, *положить*, *пуст*,
исчерпан := *с*, *переполнен*
конинт

При этом в генераторе модуля требуется указать значение *фактического параметра*:

конст *с # // стек* = (*// рстек, недопустимыйаргумент*)

Чаще используются параметры реализации. Например, в описании реализации модуля *стек* естественно ввести параметр, задающий максимальную длину стека:

```
реал //стек (длстека)
  начало
    ф64 с: = лок вект [длстека] ф64
  ...
  конец
```

Фактический параметр реализации указывается в генераторе модуля:

конст *с # //стек* = (*//рстек*, 1000)

1.31.4. Формальный контекст. Язык Эль-76 позволяет описать контекстные связи между модульными объектами, т.е. структуру программы, когда один модульный объект должен существовать всегда в контексте другого модульного объекта с заданным интерфейсом; например, когда каждая компонента большой программной системы функционирует в глобальном окружении, заданном в виде интерфейса модуля:

```
интерфейс %--- глобальное окружение --- %
    ф64 глобал1, ... , глобалк
конинт
```

Пусть *//глоб* – внешнее имя кода интерфейса глобального окружения (ядра) системы, *//рглоб* – внешнее имя кода его реализации. Интерфейс модуля-компоненты, работающего в контексте ядра, должен быть описан в следующей форме:

```
контекст (//глоб)
  интерфейс
    ф64 а, в, с: = глобал1 % используется интерфейс ядра
  конинт
```

Пусть *//инт* – внешнее имя кода интерфейса модуля-компоненты, *//реал* – его реализации. При генерации модуля с интерфейсом *//инт* должен указываться его *фактический контекст* – ссылка на экземпляр модульного объекта с интерфейсом *//глоб* (ядра системы). Таким образом, если система функционирует в контексте ядра, то последнее сначала должно быть создано как модульный объект, а затем его указатель должен передаваться как фактический контекст в генератор модуля-компоненты:

```
конст ядро # //глоб = (//рглоб); % создание ядра
конст м # //инт = (//реал, ядро); % создание компоненты
м.а: = 0; % обращение к интерфейсу компоненты
```

1.31.5. Дополнительные возможности.

О п и с а н и е   м о д у л ь н о й   п е р е м е н н о й. Модульный объект может подключаться к программе с помощью специальной формы описа-

ния переменной:

перем *c# //стек*: = (*//рстек*)

Описание модуля-константы реализуется более эффективно (интерфейс размещается в области локальных данных). Описание модульной переменной позволяет работать с модулями более свободно, например, присваивать модули. Например, если модульная переменная *c1* описана без начального присваивания (т.е. без реализации),

перем *c1# //стек*

то реализация модуля *c1* может быть задана оператором присваивания:

c1: = *c*

или

c1: = **ген** *//стек* **иниц** (*//рстек*)

Указание области памяти для интерфейса и реализации. Если *u* – вектор, достаточный для размещения интерфейса (интерфейс занимает $k + 2$ слов памяти, где k – число переменных и динамических констант интерфейса в форме **ф64**); *p* – вектор, достаточный для размещения реализации, то новый экземпляр модуля может быть создан в указанных областях памяти:

конст *u* = **лок вект** [8] **ф64**, % памяти для интерфейса
p = **лок вект** [4] **ф64**; % память для реализации
конст *c# //стек* = (*//рстек, интпам*: *u, внутрпам*: *p*)

Здесь *интпам* и *внутрпам* – стандартные имена атрибутов модуля. По умолчанию интерфейс размещается в области локальных данных (**конст** *c*) или в глобальном массиве (**перем** *c*), реализация – в глобальном массиве.

Описание модульного типа. Интерфейс и реализация могут быть описаны в одной программе. Для этого интерфейс должен задаваться *описанием типа*:

начало . . .
тип *стек* = **интерфейс ф64** *взять* . . . **конинт**

В описании реализации для обозначения интерфейса используется введенный идентификатор типа *стек* (вместо внешнего имени *//стек*):

конст *рстек* = **реал** *стек* **начало** . . . **конец**

Значение константы *рстек* – указатель реализации. Он равноправен со ссылкой в виде внешнего имени:

перем *c2# стек*: = **ген** *стек* **иниц** (*рстек*)

В данном примере создан модуль *c2* с интерфейсом *стек* и реализацией *рстек*.

Отметим, что всюду при описании модульных объектов ссылка на интерфейс, по существу, играет роль *идентификатора типа* модульного объекта, так как определяет состав и структуру интерфейса. В описании модульного типа **тип** эта роль выражена явно. Понятие модульного типа в языке Эль-76 близко понятию абстрактного типа данных.

С о з д а н и е   и н т е р ф е й с а   б е з   р е а л и з а ц и и. Генератор модульного объекта позволяет создавать экземпляры интерфейса без реализации. При этом реализация может быть даже вообще не описана. Такая возможность удобна для представления *структур* (записей) с помощью модульных объектов:

```
% описание типа структуры "комплексное число" %
тип компл = интерфейс ф64 re: = 0,im: = 0 конинт;
% описание структурных переменых типа компл:
перем z1 # компл: = ген компл,
        z2 # компл: = ген компл,
        z3# компл;
% использование структурных переменных: %
  z3: = z2;
  z1.re: = z2. re + 1; z1.im: = 2;
  запф (печ, z1.im: "z1 =" g(5),z1. re: "*i +" g(5))
% будет напечатано: z1 = 2 *i + 1
```

Средства модульного программирования играют очень важную роль в развитии программного обеспечения МВК "Эльбрус". Они использованы при описании системных технологических пакетов ИНТЕРТЕКСТ, ИНТЕРФОК, ИНТЕРКОД и ИНТЕРПАМ, что обеспечило удобную форму обращения к этим пакетам. В виде описаний модулей запрограммированы системы динамической поддержки в трансляторах. Модульные объекты языка Эль-76 удобны для описания и использования любых пакетов программ как системного, так и прикладного назначения.

1.32. Определяемый синтаксис

Любая программа в языке Эль-76 может быть вызвана как процедура с определенным набором параметров (пп. 1.30.1, 1.30.3). Однако для практической работы программиста, связанной с необходимостью использования многих программ в одном пакетном задании или сеансе диалога, наличие лишь одной процедурной формы обращения может оказаться неудобным. Основные неудобства связаны с позиционной формой параметров: она не позволяет опускать параметры, задавать для них мнемонические имена при вызове, использовать значения по умолчанию. Например, транслятор Эль-76 – это процедура-функция, имеющая определенный набор параметров, причем при обычной форме обращения необходимо помнить, что ссылка на исходный текст должна быть первым параметром, ссылка на ФОК – третьим и т.д.

Определяемый синтаксис [54] – это совокупность средств *синтаксического расширения* языка Эль-76 для создания простых *командных языков* управления системными и пользовательскими программами. Определяемый командный язык может быть использован как в пакетном, так и в диалоговом режиме. Он состоит из *директив*, каждая из которых является мнемонической формой вызова какой-либо программы и имеет вид

имя $k_1 \ldots k_n$,

где *имя* – имя директивы, k_i – параметры в мнемонической форме.

По синтаксису определяемый командный язык должен принадлежать классу $LL(1)$, т.е. для любой его директивы может быть выполнен детерминированный синтаксический анализ "по текущему символу". Форма директивы определяется *синтаксическим описанием*, в котором устанавливается соответствие между мнемонической формой вызова программы, которой соответствует директива, и обычной процедурной формой ее вызова с позиционными параметрами. Использование директив осуществляется с помощью *синтаксических подстановок*, которые могут быть заданы в качестве оператора или выражения в программе на Эль-76. По синтаксической подстановке компилятор Эль-76 генерирует код вызова соответствующей программы.

Наиболее удобная форма определения командного языка – интерфейс модуля (п. 1.31), в который входят синтаксические описания директив.

Пример.

Пусть имеется программа с внешним именем *ксн//паскаль//посл*, которая выполняет над заданной последовательностью чисел заданные действия (вычисление максимума, минимума или суммы) и выводит результаты в заданный файл. Допустим, что процедурная форма вызова этой программы имеет следующий вид:

р(*коп*, *рез*, *х1*, . . . , *хk*),

где *коп* – код операции (0 – максимум, 1 – минимум, 2 – сумма); *рез* – ссылка на файл для выдачи результата; x_i – последовательность чисел. Значение *к* может быть произвольным (программа имеет переменное число параметров).

Определим командный язык работы с этой программой, учитывающий возможность вывода на терминал:

интерфейс (э)
  синт *посл* =
    р (("max": 0, "min": 1, "сумма": 2),
      "рез" "=" ("ацпу": *читатрзадачи* (*виртвыв*),
          "экран": *позп* (э)),
      "числа" "=" "("*N*: (⟨число⟩ [", "] ") ::
    ксн//паскаль//посл
конинт

Описание интерфейса имеет параметр э (терминальный файл). Единственной компонентой интерфейса является *синтаксическое описание* (в реальных командных языках синтаксических описаний может быть до нескольких десятков). Левая часть описания состоит из слова **синт** (синтаксис) и имени директивы (*посл*). Правая часть до символа :: содержит описание мнемонической формы параметров и ее соответствия позиционным параметрам программы. Имя программы указывается в конце, после символа :: .

Самая внешняя часть синтаксического описания – *расстановка*, конструкция вида *р*(*а1*, . . . , *аk*), где a_i – альтернативы. Она обеспечивает расстановку значений параметров интерпретирующей программы, задаваемых альтернативами a_i, в порядке их описания: $a_1, \ldots, a_k$. Каждая альтернатива может задавать более одного параметра программы. При этом

мнемонические формы альтернатив, определяемые альтернативами a_i, в самой директиве *посл* могут задаваться в произвольном порядке.

Первая альтернатива расстановки ("max": 0 : "min": 1, "сумма": 2), соответствующая первому фактическому параметру, называется *ветвлением* и задает в качестве значения параметра значение одной из своих компонент, в зависимости от формы директивы. Если в ней задан ключ "max", в качестве фактического параметра передается 0, "min" – 1, "сумма" – 2.

Вторая альтернатива описывает мнемоническую форму второго параметра: последовательность *рез=ацпу* или *рез=экран*. В первом случае в качестве фактического параметра передается позиционная переменная на файл виртуального вывода, во втором – позиционная переменная на терминальный файл.

Третья альтернатива определяет мнемонику задания параметров-чисел: *числа* =(*cl*, . . . , *ck*); конструкция *N*: (⟨число⟩), (*итерация*) обозначает список чисел, разделенных запятыми; ⟨*число*⟩ – понятие синтаксиса Эль-76. Двоеточие после *N* означает, что в интерпретирующую программу передаются только сами числа (если двоеточия нет, то перед числами передается дополнительный параметр – их *количество*).

Предположим, что описание интерфейса транслировано в файл с внешним именем *ксн*//*паскаль*//*иняз*. Использование директивы *посл* осуществляется следующим образом.

1. Использование в программе на Эль-76.

пользователь = **проц** (э) % терминальный файл
  контекст конст *язык* # *ксн*//*паскаль*//*иняз* =
      (*ксн*//*паскаль*//*иняз*, э); *язык*
  начало
   посл мах числа = (1, 15, 6) *рез* = *ацпу*;
   % процедурная форма: *р*(0, *читатрзадачи* (*виртвыв*), 1, 15, 6)
   % вывод на АЦПУ: 15
    посл сумма рез = *экран числа* = (1, 2, 3, 4);
   % процедурная форма: *р*(2, э, 1, 2, 3, 4)
   % вывод на экран: 10
  нич % пустой оператор Эль-76
  конец

2. Использование в диалоговом сеансе (п. 1.33). Для входа в нестандартный диалог, совокупность директив которого определяется синтаксическими описаниями из интерфейса *ксн*//*паскаль*//*иняз*, необходимо выполнить директиву стандартного диалога:

диал ксн//*паскаль*//*иняз* (э):

где ":" – символ приглашения, выдаваемого при нестандартном диалоге. Пример диалога с использованием директивы *посл*:

:*посл* min *числа* = (60, 11, 3) *рез*=*экран*
3
:*посл сумма рез*=*экран числа* = (7, 8, 9)
24

Средства определяемого синтаксиса использованы в системе "Эльбрус" для определения основных диалоговых командных языков – *языка стандартного диалога* (п. 1.33) и *языка администратора системы.*

1.33. Язык стандартного диалога

Диалоговый режим является основным при работе программиста на МВК "Эльбрус". Работа в диалоге осуществляется на командном *языке стандартного диалога* – расширении языка Эль-76 либо на командном языке той диалоговой программы, которую использует программист. Содержание этого параграфа, вместе с описанием особенностей терминальных файлов (п. 1.26), можно рассматривать как практическое руководство по работе на МВК "Эльбрус" в диалоговом режиме. Изложение ведется применительно к терминалу ЕС-7920, наиболее распространенному в настоящее время на МВК "Эльбрус".

1.33.1. Классификация терминалов. В системе "Эльбрус" терминалы подразделяются на *операторские* и *пользовательские.* С операторского терминала можно выполнять более широкий набор директив. Иногда (например, для установки носителей на устройства) его использование необходимо (оно осуществляется оператором системы). Однако работа на нем менее удобна, чем на пользовательском терминале, так как на операторский терминал часто выдаются различные сообщения системы. На операторском терминале основным режимом работы является режим "*по активности*", при котором перед набором очередной директивы требуется нажать клавишу "ввод". На пользовательском терминале основной режим – "*по приглашению*" (стандартное приглашение системы – ">"). Пользовательский терминал может быть превращен в операторский приказом оператора:

прз пп опер,

где *пп* – номер терминала. Директива оператора

прз пп пол

превращает операторский терминал в пользовательский. В дальнейшем будет идти речь в основном о пользовательских терминалах. В конце параграфа приведены наиболее важные директивы оператора.

1.33.2. Вход в диалог. Для входа в диалог следует включить терминал и нажать клавишу "ввод". Система выдает запрос:

ваше имя?

Следует набрать ваше *имя пользователя*, зарегистрированное администратором системы. Если программист его не имеет, рекомендуется воспользоваться стандартным именем *стандпол.*. Дополнительно система может запросить *пароль*, если он установлен данным пользователем.

После правильного указания имени и пароля пользователя система выдает приглашение ">" – признак готовности к выполнению очередной директивы.

1.33.3. Установка контекста. Работа в диалоге происходит всегда в некотором *текущем контексте*, который задается внешним именем справоч-

ника в директиве *кн* (*контекст*), например:

кн ксн//лгу

Здесь *ксн//лгу* – имя базового справочника контейнера на диске. По умолчанию система устанавливает ***контекст пользователя***, хранящийся в его досье, а если он не был установлен – стандартный контекст *глобарх* (п. 1.23.2). Работа может вестись и без контекста, если он не установлен при создании досье. Справочник текущего контекста имеет в языке диалога стандартное имя – *кн*.

Директива *кн* (без параметров) выдает текущий контекст на терминал в форме:

ваш контекст ксн//лгу

1.33.4. Создание и удаление файлов и справочников. Создание файла в заданном текущем контексте выполняется директивой:

соз ф

где *ф* – имя нового файла в справочнике текущего контекста. Файл создается в том же контейнере, в котором находится справочник, и ссылка на него записывается в справочник под указанным именем. При создании файла могут быть заданы атрибуты. Например, если известно, что размер файла будет заведомо небольшим, рекомендуется при создании файла указать малое значение атрибута *длинлиста*:

соз ф [*длинлиста*: 8]

Созданный файл в примере получает внешнее имя *ксн//лгу//ф*. По умолчанию тип файла – произвольный текст, длина листа – 32 сектора.

Новый справочник создается директивой *ссп*, например:

ссп иванов

После создания личного справочника рекомендуется установить его контекст:

кн ксн//лгу//иванов

и в дальнейшем работать в этом контексте.

Удобна также следующая форма директивы *соз*, создающая новую ссылку на уже существующий объект, например, из другого справочника:

кн ксн//паскаль//тесты
соз код = ксн//паскаль//код

Для удаления ссылки на файл или справочник *//ф* служит директива:

уда //ф

Если на объект была только одна ссылка, то вместе с ней удаляется и сам объект, иначе – только ссылка (элемент справочника). При удалении справочника выдается предупреждающее сообщение (ответом должна быть пустая посылка).

1.33.5. Контроль и перепись файлов. В начале работы рекомендуется убедиться в исправности носителя и корректности информации. Директ-

тива *ктф*, например

ктф ксн//лгу//иванов

выполняет *контроль текстовых файлов* из указанного справочника (либо контроль заданного текстового файла). При ошибке в структуре файла (невозрастающей нумерации строк и т.п.) на терминал выдается соответствующее сообщение об ошибке. Чаще всего это связано с работой внешних устройств, поэтому при появлении сообщений рекомендуется переставить контейнер на другое устройство.

Перепись файла //ф1 и файл *//ф2* выполняется директивой:

пс //ф1 //ф2

Если файла *//ф2* еще нет, то он создается. Если файл *//ф2* уже существует, то его содержимое заменяется содержимым файла *//ф1*. Если *//с1* и *//с2* – справочники, то директива

пс //с1 //с2

выполняет пересылку всех файлов из справочника *//с1* в справочник *//с2* при этом файлы с именами, отсутствующими в *//с2*, создаются, о чем выдаются сообщения. Этот вариант директивы *пс* удобно использовать для дублирования.

1.33.6. Распечатка файлов и справочников выполняется директивой

печ //ф

где *//ф* – внешнее имя файла или справочника. Текстовые файлы печатаются постранично; для справочников печатаются имена файлов и их основные атрибуты. Если *//ф* – файл объектного кода программы, то печатается оглавление программы – имена всех процедур, их уровни, номера сегментов и диапазоны строк в исходном тексте.

1.33.7. Редакция текстовых файлов. В системе имеется *экранный редактор*, вход в который осуществляется директивой

ред //ф

По умолчанию (при нажатии клавиши "ввод") выполняется просмотр файла. При просмотре файл можно корректировать с помощью курсора (коррекция номеров строк не допускается). Экранный редактор имеет набор директив, каждая из которых задается одним символом и может быть набрана в любом месте экрана. После набора директивы следует нажать клавишу "ввод".

Имя директивы	Действие
л	просмотр от начала к концу
_ (подчерк)	просмотр от конца к началу
– (минус)	сдвиг вверх на размер экрана
+	сдвиг вниз на размер экрана
<	сдвиг вверх на одну строку
>	сдвиг вниз на одну строку
в	ввод новых строк после текущей строки
с	стирание текущей строки

Имя директивы	Действие
р	раздвижка текущей строки (вставка под курсор пробела)
у	удаление символа под курсором
з	запоминание текущей строки
д	дублирование запомненной строки после текущей
э	переход в диалоговый редактор
/символы/	поиск заданной последовательности символов
к	конец редакции

Текущей строкой считается строка, на которую установлен курсор. При вводе новых строк выдается приглашение в виде номера строки (номер при вводе можно корректировать, например, номер 01850100 заменить номером 01850001; при этом автоматически изменяется шаг нумерации новых строк). Конец ввода – выполнение какой-либо другой директивы, отличной от директивы *в*.

Для вставки в текущую строку символа, отличного от пробела, следует выполнить директиву *р* и затем набрать требуемый символ вместо пробела.

Разрешается запоминать директивой *з* одну или несколько строк и затем дублировать их после текущей строки. При этом строки запоминаются в порядке выполнения директивы *з* и на старом месте сохраняются. Дублируются все запомненные строки; один и тот же фрагмент может дублироваться несколько раз в разных местах файла. Любая директива (кроме *с, э, з, д* и *к*) после однократного выполнения запоминается и затем выполняется по умолчанию. При выполнении директивы *к* осуществляется выход из редактора и выдается запрос о необходимости модификации файла, на который следует ответить пустой посылкой (при этом исходный файл модифицируется; при любом другом ответе результат экранного редактирования пропадает, что удобно использовать в случае ошибочной редакции). Для более надежной работы рекомендуется использовать другую форму вызова экранного редактора – через *диалоговый редактор.*

Диалоговый редактор применяется для *построчного* и *контекстного редактирования. Вход* в него осуществляется директивой *ред//ф* и выполнением директивы э в экранном редакторе.

Приглашение диалогового редактора – символ ":". В директивах редактора принято опускать младшие нули номеров строк (например, 00125 вместо 00125000).

З а м е н а   и   с т и р а н и е   к о н т е к с т а. Директива *з*, например

з 013–014, 0256, 035 – */номер/ном/*:

заменяет в строках из указанных диапазонов заданные последовательности символов (в примере – последовательность символов "номер" на последовательности символов "*ном*"). Запись 035 означает замену во всех строках от указанной до конца файла. Двоеточие задает режим замены всех

вхождений, его отсутствие — замену только первого вхождения в каждую строку, конструкция *m* : *n* (например, 1 : 2) вместо двоеточия — замену не более чем *n* вхождений начиная с *m*-го.

Стирание контекста осуществляется директивой *с*, например:

с 013–014, 0256, 035 – /*номер*/:

В строках из указанных диапазонов стираются все вхождения заданной последовательности символов.

В с т а в к а, с д в и г и с т и р а н и е с т р о к. Последовательность

в 00005+10/
⟨вставляемый фрагмент⟩
/

вставляет указанный фрагмент (последовательность строк) начиная со строки 00005, с шагом нумерации 10. Разделитель / может быть заменен другим символом, например \ или ! Набор разделителя /, заданного в директиве вставки в первой позиции очередной строки (после номера, выдаваемого в качестве приглашения), является признаком конца ввода.

Вариант этой же директивы

в ф1 311–321 *с* 00005+1

выполняет вставку строк файла *ф1* с номерами из диапазона 311–321 в редактируемый файл. Имя *ф1* берется из текущего контекста *кн*. Если имя *ф1* не задано, выполняется дублирование строк из текущего файла.

Сдвиг строк на новое место (с их удалением на старом месте):

сдв 311–321 *с* 00005+1

На новом месте строки получают номера от 00005 с шагом 1 (если в файле уже имеются строки с такими номерами, то выдается сообщение об ошибке, и директива не выполняется).

Стирание строк:

с 311–321

П е р е н у м е р а ц и я строк из диапазона 012–013 от номера 05 с шагом 100:

нум 012–013 *от* 05+100

Если диапазон опущен, перенумеровывается весь файл. Если при перенумерации порядок номеров в файле нарушается, это считается ошибкой.

П о и с к и з а м е н а. Поиск контекста выполняется директивой *п*, например:

п 01–02 /*p* (/

В данном примере на экран при каждой пустой посылке выдается очередная строка, содержащая вызов процедуры *p*. Если необходимо во всех вызовах процедуры *p* заменить параметр-значение *x* на параметр-имя, то следует воспользоваться директивой "*поиск с заменой*":

пз 01–02 /*p* (/*x*/_*имя x*/:

З а м е н а и с т и р а н и е п о з и ц и й в с т р о к е. Пусть требуется убрать номера строк в позициях 72–80 каждой строке (такая задача часто возникает при переносе файлов с ЕС ЭВМ). Для этого служит следующий вариант директивы *с*:

с 0– *к* 72–80

Замена номеров строк на пробел выполняется директивой *з*:

з 0– *к* 72–80 //

В с т а в к а и з а м е н а о д н о й с т р о к и. Неявная директива:

00005001 _*все*

при отсутствии строки с указанным номером вставляет ее, а при наличии заменяет новым содержимым.

В ы в о д ф а й л а н а т е р м и н а л. Директива:

л 001

выводит содержимое файла на терминал начиная с указанной строки. Выводится новое содержимое файла; вставленные строки помечаются звездочками.

П е р е х о д в э к р а н н ы й р е д а к т о р, как и переход из экранного в диалоговый, выполняется директивой

э

В результате работы диалогового редактора создается рабочий файл в системном контейнере. Если экранный редактор вызывается без перехода в диалоговый, то все изменения после выхода из редактора делаются в исходном файле. Поэтому рекомендуется сеанс экранного редактирования начинать директивами

ред//ф
:э

При этом рабочий файл сохранится в системном контейнере после выхода и из диалогового, и из экранного редактора.

К о н е ц р е д а к ц и и – директива
к

При завершении редакции полностью выполняются все директивы (включая вставки и перенумерации), формируется рабочий файл на барабане и выдается сообщение об объеме файла, после чего осуществляется выход в монитор.

Для переписи результата редакции в исходный файл (сохранения файла) необходимо выполнить директиву

сох,

для переписи его в другой файл (*//ф1*) – директиву

сох ф1.

Другие формы вызова редактора:

ред //ф, //ф1 – редакция с автоматической записью результата в файл *//ф1*;

ред // *ф*, //*ф1*, //*фред* – редакция под управлением файла директив //*фред* (рекомендуется при больших редакциях). //*фред* – обычный текстовый файл, который, в свою очередь, может быть получен в результате работы редактора.

1.33.8. Трансляция. Транслятор Эль-76 – это процедура – функция с 6-ю параметрами:

автокод (*текст, втор, код, рфок, биб, сл*)

где *текст* – файл исходного текста; *втор* – *вторичный источник* (дополнительный файл неявной редакции текста, строки которого транслируются вместо строк исходного текста с совпадающими номерами); *код* – файл для объектного кода; *рфок* – контейнер для записи дополнения к файлу объектного кода; *биб* – справочник имен библиотечных карт; *сл* – файл для генерации файла ссылок, по которому выполняется распечатка словаря идентификаторов (0.1.4). Обязательны первые три параметра; по умолчанию *втор* = 0. *Результат* – *число ошибок* в программе.

Ключевая форма запуска транслятора в диалоге имеет вид

тр //*текст и2* //*втор к* //*код* > *ксн*//*рфок* (//*биб*) *сл*//*сл*

где ключ *и2* задает вторичный источник (//*втор*), ключ *к* – файл объектного кода, ключ > – контейнер для записи ДФОК; в скобках указан справочник для библиотечных карт; ключ *сл* задает файл ссылок.

Порядок всех параметров директивы произволен. Некоторые из них могут отсутствовать (в примерах разъяснены принятые умолчания).

Примеры.

1. *тр* //*текст к* //*код*

– трансляция заданного текста в заданный файл кода. По умолчанию ДФОК создается в одном контейнере с кодом; справочник библиотечных имен – текущий контекст (*кн*); *и2* = 0. Если при трансляции обнаружены ошибки, на терминал выдается предупреждающее сообщение (диагностика ошибок выводится в файл виртуального вывода).

2. *тр* //*текст синт*

– трансляция только с целью обнаружения ошибок (без последующего исполнения). Если опущено имя файла кода, то полученный код становится недоступен.

3. *тр* //*текст к раб*

– трансляция в *рабочий файл* в системном контейнере. Здесь *раб* – стандартное имя рабочего файла, который должен быть предварительно создан директивой *раб*.

4. *тр э к* //*код*

– трансляция текста, вводимого с терминала. Система выдает запрос "Вводите текст на Эль-76" и приглашение – пробел. Текст вводится до логического конца программы – последней закрывающей скобки. Для вывода сообщений транслятора на терминал рекомендуется перед текстом программы набрать следующий прагмат:

¤ *VT*

Управление трансляцией производится с помощью прагматов, вставляемых в транслируемый текст. Прагмат задается отдельной строкой текста и начинается символом "¤". Наиболее употребительные из них:

1.¤ *уст л к а си ¤*

– включение различных режимов распечатки: *л* – исходного текста, *к* – объектного кода, *а* – адресных пар (относительных адресов) локальных переменных, *си* – информации о сегментации программы, ¤ – управляющих карт. Отключение режима *л* выполняется управляющей картой:

¤ *уст*0 *л*

(аналогично – других режимов).

2. ¤ *биб //ф*

– *библиотечная карта:* переключение на трансляцию текста из файла *//ф*, по окончании которого продолжается трансляция исходного текста. Внешнее имя *//ф* понимается в контексте, который задается в директиве *тр* в виде имени справочника в скобках. По умолчанию подразумевается текущий контекст. Возможна и более полная форма внешнего имени, например:

¤ *биб ксн//паскаль//интеркод//текстквк*

3. ¤ *уст*0 *кнтрпрхд уст кормасс*

– отключение контроля локальных переходов из выражений (подавление лишней диагностики) и установка режима оптимальной трансляции операции **длина** (при условии, что длина всех массивов не превосходит $2^{16} - 1$).

Перетрансляция. Если длина программы велика (несколько тысяч строк и более), рекомендуется при исправлении локальных ошибок в процедурах выполнять не полную трансляцию программы, а *перетрасляцию* исправленных процедур. Допустим, что тексты процедур *монитор* и *сервис*, в которые внесены локальные изменения, находятся соответственно в файлах *//глоб* и *//серв*, а объектный код всей программы – в файле *//код*. Для перетрансляции и замены объектного кода процедур *монитор* и *сервис* необходимо выполнить следующие директивы:

тр э и2 //код

(выдается приглашение для ввода текста с терминала)

¤ *VT*

птрэ монитор://глоб, сервис://серв!

В результате на терминал будет выдана служебная информация о ходе перетрансляции и сообщение о ее завершении. Элементы задания на перетрансляцию (имена процедур и файлов) разделяются запятыми; признак конца задания – символ "!" – обязателен.

Трансляция с других языков. Для запуска других компиляторов, соблюдающих соглашение о порядке и смысле параметров,

описанное выше, предусмотрена следующая форма директивы *тр*:

тр //текст к //код ://транс

где *//транс* – имя кода транслятора. Для Алгола-60 и Фортрана введены особые формы:

тр //алгтекст к //код :алг
тр //форттекст к //код :форт

1.33.9. Работа с файлом виртуального вывода. Файл виртуального вывода задачи имеет в диалоге стандартное имя *ацпу*. Содержимое файла (протоколы работы системных программ и результаты счета) выводятся на терминал директивой

л ацпу

Пустая посылка в ответ на приглашение "?" – продолжение выдачи на терминал, другой ответ – окончание. В директиве можно указывать номер начальной строки:

л ацпу 00005 –

Директива:

печ ацпу

печатает на *ацпу* и стирает содержимое файла виртуального вывода.

Директива:

уда ацпу

удаляет содержимое файла виртуального вывода без его распечатки.

1.33.10. Запуск программ на исполнение осуществляется директивой

исп//код

или

исп//код (*a*1, . . . ,*ak*)

где *// код* – ссылка на файл объектного кода, a_i – фактические параметры программы. В качестве фактических параметров разрешены:

– *константы*, например: 0, "авс", пусто64, 3.14, стр8 "сообщение"
– *внешние имена*, например: *//т, ксн//паскаль//данные*
– *стандартные имена*, например: *кн, э, мойтерм*

Стандартное имя *э* (*мойтерм*) обозначает *терминальный файл задачи* и служит для передачи его в качестве параметра диалоговым программам (п. 1.26).

Программа запускается как независимый процесс (п. 1.30.2). Номер процесса в форме *:пп* выдается в системном сообщении о запуске процесса.

Параллельно с исполнением программы (как и параллельно с трансляцией большой программы) можно выполнять любые другие директивы. Не рекомендуется лишь обращаться к файлу виртуального вывода, так как он формируется в процессе исполнения. О завершении исполнения система выдает сообщение с указанием номера процесса.

При динамической ошибке в программе на терминал выдается диагностика динамической ошибки в терминах исходного текста. Система динамической диагностики имеет различные режимы выдачи символьной распечатки стека на терминал. Для управления ее работой система выдает приглашение "?". Рекомендуется в ответ набрать следующие режимы:

т тт печ

где *т* – распечатка фрагмента исходного текста, вызвавшего прерывание, с пометкой ошибочной строки; *тт* – мнемоническая распечатка кода, близкая к исходному тексту (для уточнения места ошибки в строке); *печ* – выдача информации в файл виртуального вывода. Будут распечатаны имена и номера строк вызванных процедур, имена и значения их локальных данных, а также операнды выражения, вычисление которого из-за прерывания не было завершено. Эта информация, как правило, дает возможность быстро обнаружить ошибку. На повторное приглашение "?" следует ответить "*к*" (конец диагностики). Другие режимы системы динамической диагностики можно узнать, если в ответ на приглашение "?" набрать директиву "*и*" (информация).

1.33.11. Управление процессами. В диалоговой задаче может быть запущено несколько процессов. Номер процесса в задаче имеет вид *:пп*, где *пп* – целое число. Директивы управления процессами позволяют приостанавливать процесс, выдавать распечатку его стека в терминах исходного текста, уничтожать, вновь запускать и т.п.

Директива

ост :пп

приостанавливает процесс *:пп*. Выдача символьной распечатки стека процесса выполняется директивой

л :пп

Вновь активизировать процесс *:пп* позволяет директива

ак :пп

Такая последовательность директив рекомендуется при подозрении на зацикливание программы. Директива

уда :пп

уничтожает процесс *:пп* (при этом выдается символьная распечатка его стека).

1.33.12. Информационные директивы. Информация о состоянии задачи, имеющихся в ней процессах, времени их счета или причине приостановки выдается по директиве

сз

Краткую информацию о состоянии всех задач в системе можно получить по директиве

с

информацию обо всех активных задачах (исполняемых в данный момент) – по директиве

а

С помощью этих директив можно следить за ходом исполнения своей программы и динамикой вычислений в системе, распределением оперативной и внешней памяти.

Директива

ксн

выдает на терминал информацию об установленных в данный момент контейнерах на дисках и лентах.

1.33.13. Запись дискового контейнера на ленту и восстановление его с ленты. Директива

пс ксн//паскаль мл [*эталон*]

записывает содержимое указанного контейнера на диске (*ксн//паскаль*) в контейнер на ленте с указанным именем ("*эталон*"). При выполнении директивы выдаётся трассировка в файл виртуального вывода. Обратная операция – восстановление дискового контейнера с ленты – выполняется директивой

пс мл [*эталон*] *ксн//паскаль*

1.33.14. Выход из диалога осуществляется директивой

вых

При выходе выдается предупреждение о наличии не сохраненного рабочего файла, и если на него не получено подтверждающего ответа (пустая посылка), выход не производится. При выходе диалоговая задача уничтожается, ее ресурсы освобождаются.

1.33.15. Некоторые директивы оператора. В заключение описания языка стандартного диалога приведены наиболее употребительные директивы оператора. Эти директивы могут выполняться только с операторского терминала.

Установка дисков и лент. Пусть *пп* – физический номер дисковода или накопителя на магнитных лентах. При установке на устройство нового носителя (ленты, диска) необходимо выполнить директиву

ау пп

(*ау* – активизация устройства). При этом выдается сообщение об установке на устройство *пп* контейнера с соответствующим именем.

Разметка дисков и лент. Разметка диска производится директивой

разм пп разм конт И пол П ун Н

где *пп* – номер устройства, *разм* – ключ, означающий полную разметку, *И* – имя нового контейнера, *П* – имя пользователя, *Н* – уникальный регистрационный номер. В случае разметки уже размеченного диска выдается предупреждение.

Для смены атрибута контейнера (например, имени пользователя) применяется форма той же директивы без ключа *разм*:

разм пп пол иванов

Для разметки ленты служит та же директива:

разм пп разм конт имя

Демонтирование контейнера. При снятии контейнера с устройства рекомендуется выполнить директиву *дмк*, например:

дмк ксн//паскаль,

где *ксн//паскаль* – внешнее имя контейнера.

Удаление задачи с номером *к* (например, при блокировке терминала) можно выполнить директивой

уда к

С операторского терминала можно удалять таким способом любой процесс любой задачи, например:

уда 2:5

–удаление процесса 5 задачи 2.

Управление печатью. После выключения АЦПУ в момент выдачи для ее удаления следует выполнить директиву

уда пу пп

(*пп* – номер печатающего устройства).

1.33.16. Пример сеанса диалога. Рассмотрим пример сеанса диалога, в котором выполняются наиболее часто используемые директивы – набор, редактирование, трансляция и исполнение программы. Директивы и ответы пользователя набраны курсивом, выдача системы – обычным шрифтом. Перед директивами указаны стандартные приглашения, выдаваемые системой:

```
  ваше имя?
  иванов
  мвк "эльбрус" * диалог * иванов *. дата: 5.08.87. время: 22.10
> кн ксн//лгу//иванов
> соз тест
  тест
> ред//тест
  редактор "эль". версия 3.7 от 15.06.87
  : вс 0 + 1000/
  00000000 тест = _проц (экран)
  00001000  _начало
  00002000  _ф64 а;
  00003000  а:= 1;
  00004000 запф (позп(экран), а: "а=" g(-1))
  00005000  _конец
  00006000/
  :к
```

нормальное завершение редакции
число блоков файла = 1
> *сох*
> *тр//тест к//код*
создан файл код
> *л ацпу*
(на экран выдается содержимое файла виртуального вывода)
> *уда ацпу*
> *исп//код* (э)
запущен процесс :1
а = 1
:1 процесс окончен
> *вых*
конец сеанса

ГЛАВА 2

СИСТЕМА ПРОГРАММИРОВАНИЯ ПАСКАЛЬ-ЭЛЬБРУС

При всей технологичности и универсальности языка Эль-76 средства программирования для МВК "Эльбрус" отнюдь не исчерпываются этим языком. Удобство и привлекательность новой вычислительной системы для прикладного программиста-практика состоит, по-видимому, прежде всего в наличии удобного технологического окружения, обеспечивающего работу на используемом им языке программирования (Фортран, Паскаль и т.п.). В этом отношении система программирования МВК "Эльбрус" развивается по принципу универсальности: универсальный высокопроизводительный вычислительный комплекс должен иметь универсальный спектр входных языков, удовлетворяющих запросы любой категории программистов.

В этой и двух последующих главах описываются некоторые частные системы программирования, входящие в состав программного обеспечения МВК "Эльбрус": Паскаль-Эльбрус (гл. 2), Клу-Эльбрус (гл. 3), Модула-2-Эльбрус (п. 4.6), АБВ-Эльбрус (п. 4.1), Снобол-Эльбрус (п. 4.2), Рефал-Эльбрус (п. 4.3), ДИАШАГ (п. 4.4), Форт-Эльбрус (п. 4.5).

Все эти системы создавались по общим принципам разработки системы программирования МВК "Эльбрус", описанным во введении, с применением новых методов трансляции (гл. 6). Общей чертой этих систем является также то, что все они разработаны в Ленинградском университете.

2.1. Эволюция языка Паскаль

Язык Паскаль – один из наиболее известных и широко используемых в мире языков программирования. Цель его разработки, сформулированная его автором Н. Виртом, состояла в том, чтобы "создать язык для обучения программированию как систематической дисциплине, основанной на нескольких фундаментальных понятиях, ясно и естественно выраженных в этом языке" [36]. Однако язык Паскаль благодаря своей компактности, строгости и наличию удобных конструкций для описания и обработки сложных структур данных широко применяется не только для обучения программированию, но и для разработки системных и прикладных программ. Трансляторы с языка Паскаль входят в состав базового программного

обеспечения большинства вычислительных машин. Многие понятия программирования, которые теперь относятся к числу основных, были впервые введены именно в языке Паскаль (тип-перечисление, тип-диапазон, тип-множество, оператор над записями) либо впервые выражены в нем в строгой и наглядной форме, которая послужила основой для включения в более новые языки программирования (Модула-2, Евклид, Ада и др.).

В задачу настоящей работы не входит описание основ языка Паскаль: о нем уже написано несколько сот научных и учебных работ. Достаточно назвать вышедшие в русском переводе монографии и учебники [36, 24, 74, 25]. Рассмотрим лишь некоторые особенности развития языка Паскаль, характеризующие основные проблемы его использования и реализации.

Датой создания языка Паскаль можно считать 1971 г. – год выхода статьи Н. Вирта [89]. Язык сразу привлек внимание программистов своей относительной простотой, легкостью изучения, удобством средств структурирования данных, изяществом (по сравнению, например, с языками-гигантами ПЛ/1 и Алгол-68). Однако опыт реализации и практического использования выявил, помимо неоспоримых достоинств языка Паскаль, ряд его недостатков.

1. Кажущаяся простота и небольшой объем описания языка объяснялись тем, что ряд его основных понятий не был достаточно четко определен. Это вызвало их двусмысленные толкования и поток критических статей [84]. Это относится прежде всего к известной проблеме выбора между структурной и именной эквивалентностью типов, понятиям множества, упаковки, символьной строки и некоторым другим.

2. Ощущалось неудобство использования ограниченного набора базовых конструкций структурированного программирования (операторов **if**, **case**, **while**, **repeat**)) при написании больших программ, а также недостаточность средств языка Паскаль для описания библиотек модулей, пакетов прикладных программ и абстрактных типов данных.

Вышедшие впоследствии пересмотренные варианты авторского описания языка (1973 г. – пересмотренное сообщение; 1974, 1975 и 1978 гг. – три издания книги [36]) не содержали исправлений отмеченных недостатков. В 1977 г. Н. Виртом на основе языка Паскаль создан язык модульного программирования Модула [87], в 1978 г. – его переработанный вариант Модула-2 [15]. В этих языках многие из указанных проблем решены. Однако язык Паскаль, популярность которого росла, нуждался в уточнении базовых понятий и расширении современными средствами программирования. Такая ситуация привела к тому, что все эти проблемы решались в каждой реализации языка Паскаль по-разному. В настоящее время в СССР доступно для пользователей несколько десятков отечественных и зарубежных реализаций Паскаля с различными входными языками. К примеру, в системе Паскаль-БЭСМ-6 [56] входной язык расширен операторами **select** (сокращенной формой **if**) и **branch** (со сложными передачами управления по цепочке вызовов процедур, усложняющими семантику языка), структурными метками. Система Паскаль-8000 для ЕС ЭВМ [32] имеет во входном языке описания констант сложных типов, раздел инициализации, циклы со структурными выходами, аналогичными статическим ситуациям (п. 1.15.1). Система Паскаль для СМ ЭВМ [51] дополняет вход-

ной язык функцией *loophole* для преобразования типов, функцией *ref* для вычисления адреса переменной. Наконец, новая зарубежная реализация Pascal–MT+ на ЭВМ Labtam содержит расширение языка Паскаль понятием модуля.

Уже приведенных примеров достаточно для пояснения того, что проблема стандартизации языка Паскаль и проблема учета его распространенных диалектов при реализации на новых ЭВМ весьма сложны. В 1979, 1980 и 1981 гг. группой специалистов Великобритании под руководством Т. Аддимана были разработаны три версии проекта британского стандарта языка Паскаль [80]. Четвертая версия была принята в 1982 г. в качестве британского стандарта, а в 1983 г. идентичный ему текст принят в качестве международного стандарта языка Паскаль [81]. Основные отличия стандарта от авторской версии языка [36] следующие.

1. Введены понятие *паскаль-процессора*, обобщающее все формы реализации языка Паскаль (аппаратную, компиляционную, интерпретационную и т.п.), и понятия *соответствия стандарту* и *нарушения стандарта* для паскаль-программы и паскаль-процессора. В частности, синтаксическая ошибка в конструкции языка трактуется как нарушение стандарта. Понятие *ошибки*, введенное в стандарт, близко понятию динамической ошибки (например, ошибкой считается выход за границы массива при индексации), однако учитывает и возможность статического обнаружения ошибки процессором или, напротив, отсутствия реакции на данный класс ошибок (последняя особенность процессора должна быть отражена в его сопроводительной документации).

2. Даны более строгие формулировки (в словесном виде) всем нечетко определенным в описании [36] понятиям. Формальность изложения привела к увеличению текста описания языка примерно в четыре раза и к значительным затруднениям его восприятия даже квалифицированными специалистами.

3. Введена полная спецификация параметров-процедур и параметров-функций (вместо идентификаторов [36] – заголовком).

4. Введена конструкция "*схема формальных массивов*" для облегчения программирования на Паскале процедур обработки массивов произвольной длины. Если строго следовать авторскому описанию языка [36], для умножения матриц 10 * 10 и 11 * 11 необходимо было написать две отдельные процедуры, поскольку такие матрицы описываются с помощью различных типов.

Более подробно особенности стандартного языка Паскаль пояснены в п. 2.2, где описаны основные черты входного языка системы программирования Паскаль-Эльбрус.

Появление международного стандарта несколько упорядочило решение проблемы реализации языка Паскаль на новых ЭВМ. В частности, входной язык систем Pascal-MT+ и Turbo-Pascal для зарубежных микроЭВМ является расширением международного стандарта. Планируется выпуск стандарта СЭВ на язык Паскаль и его государственного стандарта.

Стандартизация, однако, не сняла проблему различия входных языков в существующих паскаль-трансляторах. Предстоит (и частично уже выполнена) большая работа по составлению перечней отклонений входных языков от стандарта.

Можно надеяться также, что выход государственного стандарта будет способствовать решению весьма острой проблемы стандартизации русской терминологии по языку Паскаль. Синтаксис стандартного языка Паскаль приведен в приложении 1, рекомендуемая русская терминология – в приложении 2.

2.2. Принципы разработки системы Паскаль-Эльбрус

Первая версия системы программирования Паскаль-Эльбрус [64] была разработана в 1982 г. Ее входной язык соответствует авторскому описанию [36]. В качестве расширений были реализованы стандартный тип *longreal* (вещественное 128) и "большие" множества (с числом элементов до 65535), представляемые битовыми строками. Система имела хорошие эксплуатационные характеристики [64] (например, скорость трансляции на МВК "Эльбрус-1" более 12000 строк в минуту), но функционировала в первой версии только на МВК "Эльбрус-1".

Модернизация языка Паскаль (п. 2.1) и потребность в переносе системы на МВК "Эльбрус-2" привели к необходимости создания новой ее версии. Она создавалась по следующим принципам.

1. В качестве основы входного языка принят международный стандарт. Расширения стандарта связаны только с введением дополнительных предопределенных идентификаторов (констант, типов, процедур и функций), не вводят в язык новые виды операторов или описаний и не нарушают каким-либо образом лексическую и синтаксическую структуру программ, определенную стандартом.

Проблема диалектов, важная для переносимости программ с других ЭВМ, для языка Паскаль, по-видимому, не поддается удовлетворительному решению. Из примеров, приведенных в п. 2.1, видно, что для обеспечения переноса программ "на уровне колоды перфокарт" с наиболее широко используемых паскаль-систем потребовалось бы реализовать не менее пяти различных диалектов, которые частично противоречат друг другу и стандарту языка. Например, во входном языке системы Паскаль-БЭСМ-6 принята структурная эквивалентность типов, в стандартном Паскале – именная. Наиболее разумно в данной ситуации предложить программистам, желающим осуществить перенос паскаль-программ на МВК "Эльбрус", использовать только стандартные конструкции языка. Уже имеется опыт переноса нескольких программ с БЭСМ-6 и системы текстовой обработки на языке Паскаль (разработчики – Киевский филиал ИТМ и ВТ АН СССР, объем – более 10000 строк) с ЕС ЭВМ. В случае необходимости переноса очень больших программ (объемом в несколько сотен тысяч строк), для которых приведение к стандарту затруднительно, следует реализовать диалект из нескольких наиболее часто используемых расширений. Для системы Паскаль-БЭСМ-6 этот диалект составляют восьмеричные константы, оператор **select**, структурные метки и управляющие комментарии (в частности, режим отключения контроля типов).

2. При разработке использованы новые компоненты языковой поддержки ОСПО (п. 3.1.4) и расширения языка Эль-76:

– технологические пакеты ИНТЕРФОК и ИНТЕРКОД – для генерации файла объектного кода;

– форматный обмен (п. 1.19) – для реализации ввода – вывода;

– модульные объекты (п. 1.31) – для реализации системы динамической поддержки.

Использование ИНТЕР-пакетов позволило без каких-либо изменений в текстах программ осуществить перенос системы Паскаль-Эльбрус с МВК "Эльбрус-1", на котором выполнялась отладка системы, на МВК "Эльбрус-2".

3. Создание системы программирования Паскаль-Эльбрус выполнялось по новым технологическим принципам программирования – ТИП-технологии (см. гл. 5). Ее применение существенно ускорило и облегчило разработку.

2.3. Особенности входного языка Паскаль-Эльбрус

В этом параграфе описаны отличия входного языка системы программирования Паскаль-Эльбрус от более традиционной версии языка Паскаль, с которой можно ознакомиться, например, по классической работе [36]. В пп. 2.3.1–2.3.3 рассмотрены особенности международного стандарта языка Паскаль, который является основой входного языка Паскаль-Эльбрус. В п. 2.3.4 приведены расширения международного стандарта, введенные в язык Паскаль-Эльбрус.

2.3.1. Идентичность и совместимость типов. Одна из основных проблем, возникающих в языке со статической типизацией, состоит в том, какие типы следует считать идентичными (совпадающими). В частности, идентичны ли типы переменных *v*1 и *v*2:

var *v*1: **array** [1 .. 10] **of** *real*;
*v*2: **array** [1 .. 10] **of** *real*

и, в соответствии с этим, корректно ли присваивание:

*v*1 := *v*2

В языке Паскаль-Эльбрус ответ на этот вопрос отрицательный. Типы считаются идентичными в том и только в том случае, если имеет место одно из следующих условий:

– они обозначены одним и тем же идентификатором. Например: идентичны типы переменных *а* и *в* в следующем фрагменте:

type *вектор* = **array** [1 .. 10] **of** *real*;

var *a*: *вектор*; *b* : *вектор*

– они задаются разными идентификаторами, связанными соотношениями равенства через цепочку определений типов, например:

type *массив*=*вектор*;
последовательность=*массив*;
var *c* : *массив*; *d* : *последовательность*

Типы переменных *a, b, c* и *d* считаются идентичными и, следовательно, разрешены присваивания: *a* : = *c*; *b* : = *d* и т.п.

Такой подход к определению идентичности (именная идентичность) более удобен в реализации и, что не менее важно, повышает надежность

программирования, так как исключает возможность "случайного" совпадения типов, имеющих различное назначение, и, как следствие, возможность несанкционированного доступа к элементу "похожей" структуры данных. Аналогичный подход принят в языках Модула-2 и Ада. Противоположный подход (структурная идентичность), при котором типы переменных $v1$ и $v2$ считались бы одинаковыми, принят в Алголе-68.

Другой важный вопрос, связанный с типами и вызвавший много дискуссий в литературе по языку Паскаль [84], – *совместимость типов по присваиванию*: при каких соотношениях между типами переменной v и выражения e присваивание $v := e$ следует считать корректным? В языке Паскаль-Эльбрус тип выражения *e совместим по присваиванию* с типом переменной v (т.е. присваивание корректно), если:

– типы v и e идентичны (например, массив и вектор);

– переменная v имеет тип *real* выражение e – *integer*;

```
var v: real;
v: = 0
```

– переменная v и выражение e имеют простые типы, и значение e содержится в множестве возможных значений v:

```
var v: 1 .. 10;
v: = 5
```

Если в этом же примере v: = 11, то фиксируется ошибка (выход за границы диапазона);

– переменная v и выражение e имеют тип "множество", причем все элементы множества e лежат в области возможных значений элементов множества v:

```
var v: set of 0.. 20;
v: = [5, 10, 15]
```

Ошибка будет зафиксирована, если

```
v: = [0, 25]
```

– переменная v имеет тип "строка", например:

```
var v: packed array [1.. 6] of char;
```

а выражение e – строка с тем же числом элементов:

```
v: = 'строка'
```

или

```
var v1: packed array [1.. 6] of char;
v: = v1
```

Из определения следует, что присваивание переменной v сложного типа (кроме типа *строка*) возможно лишь в случаях, если тип переменной идентичен типу выражения *l*.

Во избежание недоразумений, связанных с типами, рекомендуется все используемые в паскаль-программе сложные типы обозначить идентифика-

торами с помощью определений типов, как это сделано выше для типов *массив* и *вектор.*

2.3.2. Спецификация процедурных параметров. В авторской версии языка Паскаль [36] параметр-процедура специфицируется своим идентификатором: **procedure** *p,* параметр-функция — идентификатором и типом результата: **function** *f*: *real.* Неполная спецификация процедурных параметров, унаследованная языком Паскаль от Алгола-60, приводила к нарушению принципов статического контроля типов. В международном стандарте языка Паскаль этот недостаток исправлен; процедурный параметр специфицируется своим заголовком, в котором задаются типы параметров, а для функций — и тип результата. Например, процедура *корень,* вычисляющая корень функции *f* на интервале [*a, b*], присваивающая значение корня параметру-переменной *r,* а в случае расходимости метода вызывающая процедуру-параметр *ошибка* (параметром которой является число итераций), может иметь следующий заголовок:

```
procedure  корень (a, b: real; (* концы интервала *)
           function f(x: real): real; (* функция *)
           var r: real; (* результат *)
           procedure ошибка (n: integer))
```

2.3.3. Схемы формальных массивов. Пусть *скал* — функция скалярного умножения векторов, написанная на авторской версии языка Паскаль:

```
type вектор = array [1 .. 10] of real;
function скал (var a, b: вектор): real;
var   c : real;
begin c := 0;
      for i := 1 to 10 do
          c := c + a[i] * b[i];
      скал := c
end (* скал *)
```

Эта функция может быть использована только для скалярного умножения объектов, имеющих тип *вектор,* т.е. массивов вещественных чисел с индексами от 1 до 10. Тип *вектор* должен быть, по правилам языка Паскаль, описан вне процедуры *скал,* в заголовке которой тип параметра должен специфицироваться идентификатором. Для векторов с индексами от 2 до 11 или от 1 до 100 функцию *скал* использовать нельзя, хотя тело функции зависит от индексов лишь в незначительной степени.

Для исправления этого недостатка процедур обработки массивов языка Паскаль в стандарт языка введена конструкция "*схема формальных массивов*". С использованием этой конструкции функция *скал* может быть переописана таким образом, что она становится применимой для любых векторов *a* и *в* идентичного типа с любыми целыми границами изменения индексов:

```
function  скал (var a, b : array [ l .. u : integer ] of real )
              : real;
    var c : real;
```

```
begin c := 0;
    for i := l to u do
        c := c + a[i] * b[i];
    скал := c
end (* скал *)
```

В данном примере схема формальных массивов имеет вид

array [*l* .. *u* : *integer*] **of** *real*

Идентификаторы *l* и *u* – *формальные границы* типа *integer*, обозначающие при исполнении процедуры значения фактических границ.

Их вхождения в заголовок процедуры считаются их описаниями, имеющими смысл в теле процедуры (поэтому, в частности, повторное использование идентификатора *l* или *u* в заголовке той же процедуры не допускается). Формальные границы используются в теле процедуры для указания границ изменения параметра цикла *i*.

Функцию *скал* в новом варианте можно применять к векторам *x* и *y* любого размера с целыми индексами и идентичными типами:

```
type вектор1 = array [1 .. 10] of real;
     вектор2 = array [0 .. 100] of real;
var x1, y1 : вектор1;
    x2, y2 : вектор2;
...
writeln (скал (x1,y1));
writeln (скал (x2,y2))
```

Вызов функции

скал (*x*1,*y*2)

ошибочен, так как формальные параметры *a* и *b* специфицированы *одной и той же схемой* формальных массивов и, следовательно, по правилам стандартного языка Паскаль должны иметь *идентичные типы*. Типы переменных *x*1 и *y*2 не идентичны.

Схема формальных массивов может описывать и многомерные массивы:

```
procedure умнматр
(var a: array [la1 .. ua1 : integer; la2 .. ua2 : integer] of real;
var b: array [lb1 .. ub1 : integer; lb2 .. ub2 : integer] of real;
var c: array [lc1 .. uc1 : integer; lc2 .. uc2 : integer] of real)
```

Фактическими параметрами процедуры *умнматр* могут быть, например, следующие матрицы:

```
var x : array [1 .. 5, 1 .. 10] of real;
    y : array [1 .. 10, 1 .. 7] of real;
    z : array [1 .. 5, 1 .. 7 ] of real;
    ... умнматр (x, y, z)
```

На этом примере видны и преимущества, и недостатки схем формальных массивов. С одной стороны, параметрами процедуры *умнматр* могут быть матрицы любого размера. С другой стороны, невозможно задать с помощью

схем формальных массивов согласованные размеры матриц, так как нельзя повторно использовать идентификатор границы в заголовке процедуры (следовательно, проверку согласованности размеров матриц требуется осуществлять динамически).

Согласно стандартному описанию языка Паскаль типа "массив массивов" и "двумерный массив" считаются различными обозначениями одного и того же типа. Соответственно переменная $x[i, j]$ считается сокращенным обозначением переменной $x[i][j]$. Эта особенность отражена и в схемах формальных массивов. Схема, использованная в заголовке процедуры *умнматр*, имеет следующую полную форму:

array [*l*1 .. *u*1 : *integer*] **of array** [*l*2 .. *u*2 : *integer*] **of** *real*

Схемы и строки. В языке Паскаль принято считать тип

packed array [1 .. *n*] **of** *char*

типом *строка*. Над строками определены некоторые операции, которые рассматривают их как единое целое, например, операции сравнения. Однако если тип формального параметра процедуры задан схемой упакованных массивов с типом элемента *char*:

procedure *p* (**var** *s* : **packed array** [*l* .. *u* : *integer*] **of** *char*)

то параметр *s* не может рассматриваться как строка, а его обработка должна выполняться *поэлементно*: *s* [*i*].

Схемы и многоязыковое программирование. Схемы формальных массивов играют в языке Паскаль-Эльбрус важную роль, связанную с использованием *внешних процедур* (в частности, процедур на других языках). Если в программе имеется заголовок внешней процедуры с параметрами-массивами, то последние должны быть специфицированы схемами формальных массивов:

procedure *внешняя* (**var** *a* : **array** [*m* .. *n* : *integer*] **of** *real*);

external; (* или *fortran* *)

В этом случае при передаче фактического параметра-массива в процедуру *внешняя* будет создан стандартный паспорт массива, используемый для представления массивов в большинстве систем программирования для МВК "Эльбрус". В паскаль-программе без внешних процедур массивы реализуются более эффективно (без использования паспортов), так как в языке Паскаль границы любого массива известны при трансляции.

2.3.4. Упаковка. Как известно, в языке Паскаль тип можно описать как *упакованный*:

```
type структ = packed record
                 a : array [1 .. 3] of 0 .. 5;
                 b : boolean;
                 c : packed array [1 .. 5] of char end;
var v : структ
```

Это означает, что компилятор должен максимально экономить память для представления значения переменной *v*, упаковывая значения полей

записи в минимально необходимое число битов. В авторском описании оставался неясным вопрос, на какую глубину структурирования распространяется упаковка (в частности, следует ли упаковывать элементы массива *v.a*). Более гибкое управление памятью обеспечивается при следующем подходе, который и был принят в международном стандарте языка Паскаль. Символ **packed** влияет на представление только того уровня структурирования типа, на котором он задан. Для переменной *v* это означает, что упаковывать следует значения полей записи: *v.a, v.b, v.c*, рассматриваемые как единые объекты. Элементы массива *v.a* не упаковываются, а элементы массива *v.c* упаковываются, так как признак упакованности имеет сам массив. Транслятор Паскаль-Эльбрус отводит для массива *v.a*. 3 слова, для значения *v.b* – один бит, для значения *v.c* – 5 байтов. Общая длина значения переменной *v* равна, таким образом, 4-м словам (с учетом выравнивания полей и округления общей длины до наибольшего формата). Если же тип поля *v.a* также имеет признак упаковки:

a : **packed array** [1 .. 3] **of** 0 .. 5

то упаковываются и значения элементов массива *v.a*. При этом каждый из них займет по 3 бита, весь массив – 9 битов, а длина записи *v* равна, следовательно, одному слову.

Описанный подход к упаковке имеет и недостатки, так как он не вполне согласуется с использованием двух форм обозначения типа *массив*. Например, поскольку сокращенная форма обозначения типа

array [1 .. 10, 1 .. 10] **of** 0 .. 3

считается эквивалентной *полной* форме

array [1 .. 10] **of array** [1 .. 10] **of** 0 .. 3

то обозначение типа

packed array [1 .. 10, 1 .. 10] **of** 0 .. 3

эквивалентно обозначению типа

packed array [1 .. 10] **of packed array** [1 .. 10] **of** 0 .. 3

При этом обозначение типа

array [1 .. 10] **of packed array** [1 .. 10] **of** 0 .. 3

не имеет эквивалента в полной форме.

2.3.5. Расширения. Рассмотрим расширения международного стандарта, принятые в языке Паскаль-Эльбрус. Диагностика отступлений от стандарта при использовании этих расширений может быть получена при включении специального режима транслятора (п. 2.4.3).

Дополнительные стандартные числовые типы:

– *shortint* – целое 32;

– *shortreal* – вещественное 32;

– *longreal* – вещественное 128.

Эти три типа, в совокупности с типами *integer* (целое 64) и *real* (вещественное 64), дают возможность использовать в одной программе на языке Паскаль все типы и форматы чисел МВК "Эльбрус". В частности, тип *longreal* предназначен для вычислений с двойной точностью. Все целые типы считаются совместимыми по присваиванию между собой и со всеми вещественными типами. Если при присваивании $v := e$ переменная v имеет тип *shortint*, а выражение e – тип *integer*, то присваивание считается корректным, если значение e представимо в виде значения типа *shortint*, иначе фиксируется ошибка (аналогично для *real* и *longreal*). Тип вещественной константы определяется по числу значащих цифр мантиссы:

1 – *integer*;
222222222222 – *integer*;
3.14 – *treal*;
7.256103492 – *real*;
1.23456765432123456765432l*e*–2 – *longreal*

Такая трактовка числовых констант не приводит к нарушению стандарта языка Паскаль, так как различные целые и вещественные типы считаются совместимыми.

Д о п о л н и т е л ь н ы й с т а н д а р т н ы й т и п *alfa*. Значениями этого типа считаются все короткие символьные строки-константы (длиной не более 8 символов). Иначе говоря, типа *alfa* соответствует байтовому набору системы "Эльбрус" (не обязательно полному). Например, если переменные *a*1 и *a*2 имеют тип *alfa*:

var *a*1, *a*2 : *alfa*

то им можно присваивать любые строки длиной не более 8 символов:

*a*1 := '*abc*';
*a*2 := '*строка*'

Значения типа *alfa*, как и строки, можно сравнивать. При этом, в отличие от строк, они могут иметь различную длину:

*a*1 = *a*2

*a*2 < '*стр*'

Тип *alfa* введен как из соображений удобства, так и с целью облегчения переноса паскаль-программ с БЭСМ-6. В языке Паскаль-монитор на БЭСМ-6 [56] тип *alfa* активно используется.

В о с ь м е р и ч н ы е ц е л ы е к о н с т а н т ы. Изображение константы типа *integer* восьмеричными цифрами отличается по форме от обычного десятичного изображения символом "B" после изображения числа:

100B = 64

Восьмеричные константы, как и тип *alfa*, введены для разработки системных программ и для переноса программ, написанных на языке Паскаль-монитор для БЭСМ-6.

Это расширение не нарушает корректности программ, удовлетворяющих стандарту, так как последовательность символов 100B не может встретиться в синтаксически правильной (с точки зрения стандарта) паскаль-программе.

Дополнительные стандартные функции.

Операции над множествами. Если *s* – множество, т.е. значение или переменная типа **set of** *t* или **packed set of** *t*, то:

card(*s*) – число элементов (мощность) множества *s*;

maxel(*s*) – элемент множества *s* с максимальным номером (значением стандартной функции *ord*).

Операции *card* и *maxel* дают возможность организации циклов эффективной обработки множеств без введения для этого дополнительных операторов. Например:

```
type si = set of integer; (* множество целых чисел *)
procedure writeset (s : si);
(* вывод элементов множества *)
var c, i : integer;
begin c := 0;
   while card (s) < > 0 do
      begin i := maxel (s);
         writeln (i);
         s := s – [i]
      end
end (* writeset *)
```

Несколько слов о трактовке понятия множества в языке Паскаль-Эльбрус. Тип *множество* **set of** *t* понимается как тип

set of *l* .. *u*

где *l* и *u* – значения типа *t*; *ord*(*l*) = 0; *ord*(*u*) = 255, если *t* – символьный тип (*char*), иначе *ord*(*u*) = 63. Таким образом, выделены два класса множеств: малые (с элементами типа *целое, логическое* или *перечисление*) и большие (с элементами типа *символ*). Такой подход не нарушает стандарта (который не проясняет окончательно трактовку множеств) и соответствует практическим потребностям программистов: в большинстве паскаль-программ используются именно такие классы множеств. Введенный в примере тип **set of** *integer* в языке Паскаль-Эльбрус понимается как **set of** 0 .. 63, поэтому элементами множества *s* могут быть только целые числа из диапазона 0 .. 63.

Дата и время. Функция *clock* без параметров выдает текущее значение использованного процессорного времени задачи в единицах центрального процессора (результат имеет тип *integer*). По аналогии с языком Эль-76 введена стандартная константа *second* типа *integer*, значением которой является число единиц времени центрального процессора в одной секунде.

Функции *date* и *time,* также без параметров, выдают в виде значений типа *alfa* текущую дату и текущее время, например:

date = ' 07.08.87 '   *time* = ' 16.55.20 '

Пример использования стандартных функций:

```
program замер (output);
var t, x : real; i : integer;
begin
   writeln (' дата =', date, ' время =', time);
t := clock; (* начало отсчета времени *)
for i := 1 to 1 000 000 do
      x := 1/i;
t := (clock – t)/second; (* конец отсчета *)
writeln (' время счета =', t : 7 : 2, ' сек.' )
end.
```

2.3.6. Лексика. В языке Паскаль-Эльбрус принято традиционное представление символов операций и разделителей:

– вместо символа "↑", используемого в обозначении типа "указатель" и в конструкции "косвенная переменная", применяется символ "@";

– вместо фигурных скобок в качестве ограничителей комментариев используются пары символов, (* и *). В соответствии со стандартом языка вложенные комментарии не допускаются:

(* это не (* вложенный коментарий *)   . ;

– для выделения строк используется символ "апостроф";

– для квадратных скобок можно использовать альтернативные представления (. и .). Это удобно, например, для совместимости с системой Паскаль-8000 для ЕС ЭВМ.

Р у с с к а я л е к с и к а. Для удобства программистов, которым более привычны языки с русской лексикой (например, Эль-76), в языке Паскаль-Эльбрус разрешено использование русских букв в идентификаторах и строках, введена русская мнемоника служебных слов и предопределенных идентификаторов (см. приложение 2). Отметим, что язык Паскаль до сих пор не имеет стандартного или хотя бы устоявшегося варианта русской лексики.

Использование русской и английской мнемоники для предопределенных идентификаторов (например, *sin* и *син*) допускается одновременно, хотя и не рекомендуется. Русская и английская мнемоника служебных слов исключают друг друга (переключение транслятора с одной на другую выполняется с помощью прагмата, см. п. 2.4.2). По умолчанию установлена английская мнемоника как более привычная.

Такой подход к введению русской лексики не приводит к запрещению использования (резервированию) каких-либо идентификаторов, дополнительно к списку служебных слов, зафиксированному в какой-либо мнемонике.

П р и м е р паскаль-программы в русской мнемонике:

программа *печать* (*ввод, вывод*)
   (* печать чисел *)
перем *число* : *цел*;
начало
   пока не *конф* **цикл**
      начало
         чит (*число*);
         печ (*число*)
      конец
конец.

2.4. Методы программирования и управления транслятором

В данном параграфе приведены необходимые практические сведения по использованию транслятора Паскаль-Эльбрус и рекомендации по программированию с учетом его возможностей.

2.4.1. Запуск транслятора. Как и любой транслятор в системе "Эльбрус", паскаль-транслятор может использоваться как в диалоговом, так и в пакетном режиме.

Д и а л о г о в ы й   р е ж и м. Для запуска транслятора служит одна из форм директивы *тр* языка стандартного диалога (п. 1.33.8), например:

тр // текст к // код : *ксн // паскаль // код*   ,

где *//текст* – внешнее имя текста программы, *//код* – внешнее имя ее файла объектного кода, *ксн//паскаль//код* – внешнее имя кода транслятора. Паскаль-транслятор соблюдает все стандартные соглашения о числе и порядке параметров, результате трансляции и неявно выполняемых действиях по созданию файлов (п. 1.33.8). Поэтому все варианты его запуска ничем не отличаются от вариантов запуска транслятора Эль-76, описанных в п. 1.33.8, кроме указания имени кода транслятора после символа " : ". В частности, по умолчанию транслятор генерирует дополнение к файлу объектного кода, создавая файл для него в одном контейнере с файлом объектного кода.

Если в программе обнаружены ошибки, то после завершения трансляции система стандартного диалога, как и для транслятора Эль-76, выдает на терминал сообщение:

были синтаксические ошибки!

и приглашение ">". Для анализа диагностики следует воспользоваться директивой: *л ацпу* (*печ ацпу*) (см. п. 1.33.9).

В в о д   т е к с т а   с   т е р м и н а л а осуществляется после набора директивы

тр э к // код : *ксн // паскаль // код*

В отличие от транслятора Эль-76 паскаль-транслятор выдает *приглашение в виде номера строки* (п. 1.26.1) из восьми цифр. Конец исходного

текста – пустая посылка (нажатие клавиши "ввод" сразу после выдачи номера строки). Протокол работы транслятора всегда выдается в файл виртуального вывода. Для его выдачи на экран следует вначале набрать управляющую карту ¤ *уст вт*.

Пример ввода текста с терминала (набор пользователя выделен курсивом, пустая посылка обозначена символом " ■ "):

```
《《 транслятор Паскаль-Эльбрус. версия 7.09.87 》》
00000000 ¤ уст вт
00001000 program p (output);
00002000 begin
00003000 writeln ('работает')
00004000 end.
00005000 ■
           размер программы (в строках) = 5
           объем кода (в словах)        = 10
           время трансляции             = 00.01 сек.
《《 конец работы Паскаль-транслятора 》》
```

Признаком нормального завершения ввода текста с терминала является выдача системой приглашения " >" для набора очередной директивы стандартного диалога.

П а к е т н ы й р е ж и м. В пакетном режиме, как и в диалоговом, транслятор Паскаль-Эльбрус запускается точно так же, как и транслятор Эль-76 (пп. 1.30.3, 1.33.8). В отличие от последнего, паскаль-транслятор не имеет предопределенного идентификатора и запускается по внешнему имени:

```
начало
    конст паскаль = прогр (ксн // паскаль // код);
    паскаль (ксн // лгу // текст, 0, кон // лгу // код)
конец
```

2.4.2. Прагматы. Управление режимами работы транслятора Паскаль-Эльбрус осуществляется с помощью прагматов. По форме прагматы унифицированы с Эль-76 (п. 1.33.8). По традиции реализованы также *управляющие комментарии* для наиболее часто используемых в различных паскаль-системах режимов трансляции. В тех случаях, когда это возможно, для соответствующего режима указаны обе формы управления.

Прагмат имеет вид

¤ *уст* $p_1 \ldots p_n$

где *уст* – директива установки, p_i – имена режимов. Управление большинством режимов осуществляется по стековому принципу: режим p – стек логических значений; текущее состояние режима – вершина стека; операция ¤ *уст p* помещает в вершину стека 1, операция ¤ *уст*0 *p* – нуль, операция ¤ *вос p* – восстанавливает предыдущее состояние стека. Директива ¤ *восуст p* эквивалентна последовательному выполнению директив ¤ *вос p* и ¤ *уст p* (аналогично – ¤ *восуст*0 *p*). Такая форма управ-

ления режимами заимствована из транслятора Эль-76. Ее удобство в том, что состояние режима может быть установлено на время трансляции некоторого фрагмента программы, без изменения состояния этого режима в остальной части программы:

¤*уст л*
  (* этот фрагмент текста печатается всегда *)
¤*вос л*
  (* этот текст печатается, если режим *л* был установлен раньше *)

Ниже описываются основные режимы паскаль-транслятора. Для каждого из них указано полное имя, сокращенное имя, значение по умолчанию, смысл и особенности использования. Режимы, связанные с установкой или отключением *контроля*, описаны отдельно в п. 2.4.3, *библиотечные карты* – в п. 2.4.4, *режим перетрансляции* – в п. 2.4.5, управляющие карты для использования *пакетов программ* – в п. 2.4.6.

Информационные режимы.

лист (*л*, **ложь**) – распечатка исходного текста. Управление этим режимом возможно также с помощью управляющего комментария *L*: форма (* ¤ *L*+ *) соответствует ¤ *уст л*, форма (* ¤ *L* – *) – ¤ *уст*0 *л*.

код (*к*, **ложь**) – мнемоническая распечатка объектного кода. Код печатается по процедурам; перед каждой командой указываются ее координаты в форме *сегм* : *слово* : *байт*, где *сегм* – номер программного сегмента процедуры, *слово* – номер слова в сегменте, *байт* – номер байта в слове.

адстек (*а*, **ложь**) – распечатка адресных пар локальных переменных, формальных параметров и других величин. Например:

x = / 1, 4 /

где *x* – имя переменной, 1 – *уровень*, 4 – *смещение* в полусловах.

сегинф (*си*, **ложь**) – распечатка информации о сегментации. Для каждой процедуры указывается ее имя, номер программного сегмента, уровень, диапазон строк и длина сегмента в словах.

¤ (**ложь**) – печать управляющих карт.

Режимы генерации.

дд (**истина**) – генерация дополнения к файлу объектного кода. По умолчанию транслятор генерирует ДФОК, так как это обеспечивает динамическую диагностику в терминах исходного текста, комплексацию программ и другие услуги средств языковой поддержки ОСПО (п. В.1.4). По желанию программиста генерация ДФОК может быть отключена (например, для экономии внешней памяти, если при трансляции большой паскаль-программы ее объектный код записывается в системный контейнер).

словарь (*сл*, **ложь**) – генерация файла ссылок для распечатки словаря идентификаторов (информации об их использованиях, определениях и модификациях в программе). Файл ссылок состоит из записей, каждая из которых соответствует либо одному определяющему вхождению или использующему вхождению идентификатора, либо началу или концу блока. Для создания файла ссылок транслятору необходимо передать в качестве

параметра с ключом "*сл*" некоторый файл:

тр // текст к // код : ксн // паскаль // код сл // фссылок

Режим игнорирования

игнр (*и*, **ложь**)

Предназначен для *игнорирования* фрагмента текста:

¤ *уст игнр*

этот фрагмент игнорируется (воспринимается как комментарий)

¤ *вос игнр*

Режим управления протоколом. *вт* (**ложь**) – выдача протокола на терминал. Если исходный текст вводится с терминала, то включение этого режима рекомендуется. Вывод протокола на терминал возможен и в том случае, если исходный текст задается не с терминала. Для этого следует запустить паскаль-транслятор директивой *исп* (п. 1.33.10) и передать терминальный файл седьмым параметром:

исп ксн // паскаль // код (// *текст*, 0, // *код*, 0, 0, 0, э)

Протокол выводится порциями, по размеру экрана. После выдачи очередной порции выдается приглашение " ? " (пустая посылка – продолжение выдачи, любой другой ответ – прекращение выдачи протокола на терминал).

Независимо от режима *вт* протокол работы транслятора всегда выдается в файл виртуального вывода.

Режим оптимизации. *оптим* (**ложь**) – оптимизация обращения к массивам в циклах (линеаризация массивов). Например, при трансляции фрагмента

```
¤ уст оптим
for i := 1 to 1000 do
for j := 1 to 1000 do
    a [i, j] = b[i, j] + 1
```

транслятор выполняет оптимизацию, при которой обращение к элементам двумерных массивов *а* и *в* транслируется как обращение к элементам одномерных массивов (линеаризуется). В результате время исполнения данного фрагмента сокращается приблизительно в 1,5 раза. Оптимизация выполняется с помощью технологического пакета ИНТЕРКОД.

Режим ограничения числа ошибок. Прагмат

¤ *уст максош = пп*

устанавливает максимальное число ошибок, обнаруживаемых в программе, равным *пп*. При превышении указанного числа ошибок трансляция прекращается, по умолчанию *максош* = 50.

Р е ж и м у п р а в л е н и я м н е м о н и к о й. *рус* (**ложь**) – установка русской мнемоники служебных слов. При выключении режима *рус* устанавливается английская мнемоника.

Р е ж и м о т к р ы т о й п о д с т а н о в к и. Прагмат

¤ *уст откр р*1 ... *pn* ,

где *p*1, ..., *pn* – имена процедур (функций), определенных в программе, задает режим реализации вызова этих процедур и функций *открытой подстановкой* в место их вызова. Отключение режима *откр* для процедур *p*1, ..., *pn* задается прагматом:

¤ *уст*0 *откр* p_1 ... *pn*

Этот режим может использоваться как самостоятельно (для повышения эффективности программы), так и при применении ТИП-технологии (п. 5.4.7).

Р е ж и м п р и м е н е н и я Т И П-т е х н о л о г и и (п. 5.4.7). Управляющая карта

¤ *тип*

устанавливает трансляцию программы в режиме использования ТИП-технологии. В этом режиме разрешен произвольный порядок описаний, а также автоматически включается режим *откр* для всех процедур.

2.4.3. Виды контроля и управление ими. В процессе работы транслятор выполняет различные виды контроля: контроль синтаксической правильности программы, контроль типов, контроль соответствия текста программы стандарту языка Паскаль (т.е. фиксация используемых расширений стандарта), контроль правильности структуры входного текстового файла. Кроме того, транслятор генерирует команды, выполняющие динамический контроль ошибок. Некоторые из этих видов контроля могут включаться или отключаться специальными управляющими картами транслятора.

К о н т р о л ь п р а в и л ь н о с т и с т р у к т у р ы т е к с т о в о г о ф а й л а.

ошнумвх (**истина**) – контроль возрастания номеров строк в исходном текстовом файле. Невозрастающая нумерация, как правило, вызвана либо нарушением структуры текстового файла (порчей информации), либо сбоями в работе внешних устройств. Поэтому в режиме *ошнумвх* нарушение возрастающей нумерации строк в исходном тексте приводит к прекращению трансляции.

Как и другие компиляторы в системе Эльбрус, система Паскаль-Эльбрус (в отличие, например, от интерпретаторов языка Бейсик для различных ЭВМ) не имеет возможности коррекции кода программы при повторном задании строки с одним и тем же номером. Это потребовало бы особой (пошаговой) организации компилятора (см. п. 4.4), при которой скорость трансляции значительно уменьшилась бы.

К о н т р о л ь с о о т в е т с т в и я с т а н д а р т у (*фильтр*).

станд (**ложь**) – контроль и диагностика использования расширений стандарта языка Паскаль, введенных в язык Паскаль-Эльбрус (п. 2.3.5). Если режим *станд* включен, то на каждое использование расширения вы-

дается предупреждающее сообщение. Например, для фрагмента

```
¤ уст станд
    type d = 10B .. 20B;
        a = array [ d ] of shortint
```

диагностика о расширении стандарта будет выдана три раза: дважды используется восьмеричное изображение целой константы и один раз – стандартный тип *shortint*. Управление этим режимом возможно также с помощью управляющего комментария *s*: форма (* ¤ *s* + *) соответствует ¤ *уст станд*, форма (* ¤ *s* – *) – ¤ *уст*0 *станд*.

Контроль типов.

типы (**истина**) – контроль соответствия типов при присваивании и передаче параметров. Паскаль – язык со строгой типизацией, поэтому по умолчанию режим *типы* включен. Однако, как известно, при использовании статического языка в системных программах может возникнуть необходимость ослабления статического контроля типов. Например, может оказаться удобным передать в качестве параметра в процедуру переменную-запись, а в процедуре обработать ее как массив соответствующей длины:

```
type r = record a, b, c, d, e : real end;
   ar  = array [1 .. 5] of real;
var vr : r;
procedure print (var a : ar);
var i : integer;
begin for i := to 5 do
        writeln ('элемент номер', i, 'равен', a[ i ])
end;
¤ уст0  типы
   print (vr)
```

В результате вызова процедуры *print* с отключенным контролем типов будут распечатаны значения полей записи *vr* как элементы массива. При использовании режима *типы* следует иметь в виду, что знание особенностей представления данных, соответствия размеров рассматриваемых структур и прочий контроль в этом случае возлагаются на программиста.

Управление режимом контроля типов (который является управляемым для многих паскаль-систем) может осуществляться также с помощью управляющего комментария *с*: (* ¤ *с* + *) эквивалентно ¤ *уст типы*, (* ¤ *с* – *) – ¤ *уст*0 *типы*.

Динамический контроль.

дин (*д*, **истина**)

В языке Паскаль-Эльбрус контролируется более 30 видов динамических ошибок (п. 2.5.4). При отключении режима *дин* транслятор Паскаль-Эльбрус не генерирует объектный код динамических проверок, специфичных для языка Паскаль (например, проверки выхода за границу диапазона при присваивании переменной типа *диапазон*). Поэтому режим *дин* рекомендуется отключать для отлаженных программ, для которых существенна эффективность выполнения. Весьма важно, что независимо от состояния режима *дин* динамический контроль многих распространенных классов

ошибок (например, использования неопределенного значения переменной) осуществляется аппаратно.

Управление режимом *дин* возможно с помощью управляющего комментария *т*: форма (* ¤ *т* + *) соответствует ¤ *уст дин*, форма (* ¤ *т* – *) – ¤ *уст* 0 *дин*.

2.4.4. Библиотечные карты служат для трансляции программы, текст которой разбит на несколько текстовых файлов. Файл, содержащий библиотечный текст, задается внешним именем. Как правило, каждый из библиотечных текстов содержит либо глобальные описания, либо описание процедуры, либо описания нескольких, тесно связанных по смыслу и взаимодействию процедур

```
program пример (output);
   ¤ биб // глобалы
   ¤ биб // проц1
   . . .
   ¤ биб // процN
```

Справочник-основа для библиотечных внешних имен передается транслятору в качестве параметра (в директиве *тр* он указывается в скобках) :

тр // текст к // код : ксн // паскаль / код (// бибспр)

По умолчанию основой считается текущий контекст *кн* (п. 1.33.3). При указании полных внешних имен

¤ *биб ксн // паскаль // тесты // тест* 1

основу задавать не требуется. Для удобства трансляции больших программ с библиотечными картами рекомендуется следующий способ вызова паскаль-транслятора:

```
тр э к // код : ксн // паскаль // код
00000000  ¤  уст vт ¤
00001000  ¤  биб // текст
```

где // *текст* – внешнее имя основного файла текста программы. На экран будут выдаваться сообщения транслятора и библиотечные карты транслируемых частей программы.

2.4.5. Перетрансляция. При исправлении локальных ошибок в процедурах большой паскаль-программы следует использовать режим перетрансляции. Форма задания на перетрансляцию унифицирована с транслятором Эль-76 (п. 1.33.8). Однако перетрансляция процедур на языке Паскаль имеет свои особенности. Процедура, для которой предполагается перетрансляция, должна иметь предварительное описание, например:

```
procedure поиск (var a : array [m . . n : integer] of integer;
                  x : integer; var i : integer);
                  forward';
( * предварительное описание * )
. . .
procedure поиск;
( * заголовок не повторяется; см. предварительное описание * )
```

```
var k : integer; found : boolean;
begin
k := m; found := false;
while k < = m and not found
do
   if a[k] = x then
      begin found := true; i := k end
else k := k + 1;
if not found then i := 0
end (* поиск *)
```

Предварительное и окончательное описание процедуры *поиск* для удобства рекомендуется хранить в разных текстовых файлах. Например, предварительное описание процедуры может храниться в файле глобальных описаний. Пусть *// поиск* – внешнее имя файла с окончательным описанием процедуры *поиск*, *// код* – имя файла кода всей программы. Если изменения вносятся только в локальные описания и в тело процедуры (а не в ее заголовок), то для перетрансляции процедуры требуется выполнить следующие директивы диалога:

```
тр э и2 // код
00000000 ¤ уст ит
00001000    птрэ поиск : // поиск!
```

Признак завершения перетрансляции – выдача заключительных сообщений паскаль-транслятора (см. пример к п. 2.4.1) и приглашения ">" системы стандартного диалога.

Если процедура содержит глобальные идентификаторы (например, если в процедуре *поиск* переменная *i* – глобальная), форма директив не изменяется. Смысл глобальных идентификаторов определяется по контексту, в котором описана процедура.

Если требуется изменить *заголовок* процедуры *поиск* (например, поменять порядок параметров), то перетранслировать следует объемлющую процедуру (либо всю программу, если процедура *поиск* глобальная). В этом основное отличие перетрансляции процедур языка Паскаль от Эль-76. Заголовок процедуры на языке Паскаль играет роль спецификации ее интерфейса и используется для статического контроля типов при вызове процедуры. Заголовок процедуры Эль-76 используется только при трансляции тела процедуры, так как в Эль-76 статический контроль при вызовах процедур стандартного типа отсутствует.

2.4.6. Работа с пакетами прикладных программ. Система Паскаль-Эльбрус предоставляет возможность описания пакетов прикладных программ на языке Паскаль и использования пакетов программ на Паскале или на Эль-76. Обеспечивается также возможность использования пакетов на языке Паскаль в программе на Эль-76.

Пакет прикладных программ на языке Паскаль-Эльбрус – это программа, содержащая описания процедур и функций. Программа-пакет не должна иметь параметров, глобальных описаний меток, переменных, типов-указателей, констант-строк длиной более 8 символов, т.е. описаний, приводящих

к отведению памяти в области локальных данных программы. Аналогичные ограничения накладываются на описание программы-пакета в языке Эль-76 (п. 1.18.3). Тело программы (раздел операторов) игнорируется. В процедурах пакета не должны использоваться операторы ввода-вывода (они могут задаваться непосредственно в программе-пользователе пакета). Внутри процедур, составляющих пакет, разрешены любые описания.

Программа-пакет транслируется в специальном режиме, который включается прагматом ¤ *уст пак,* заданным перед текстом программы.

Подключение пакета производится прагматом вызова пакета, содержащим внешнее имя файла объектного кода программы-пакета:

¤ *пак ксн//паскаль//кодпак*

Прагмат вызова пакета должен быть задан после заголовка паскаль-программы, до ее глобальных описаний. Никаких ограничений на программу, использующую пакет, не накладывается. В программе-пользователе доступны все константы, типы и процедуры, описанные в пакете. Каких-либо предварительных описаний процедур пакета в программе-пользователе не требуется. Если программа, использующая пакет, содержит описания идентификаторов, уже описанных в пакете, то это рассматривается как переописание.

При необходимости в прагмате ¤ *пак* можно указать список используемых процедур пакета.

Пример 1.

Описание и использование пакета на языке Паскаль. Пакет предназначен для работы с векторами.

```
(* описание пакета *)
¤ уст пак
program векторы;
(* пакет для работы с вещественными векторами *)
 function скал (var a, b: array [m.. n: integer ] of real): real;
  (* скалярное произведение *)
var i: integer; r: real;
begin r:= 0.0;
  for i:=m to n do
     r:=r+a [i] * b [i];
   скал :=r
 end(* скал *);
function max (var a: array [m..n: integer ] of real): real;
(* максимальный элемент вектора *)
var i: integer; x: real;
begin x: =a [m];
   for i: =m + 1 to n do
   if a [i] > x then x: =a [i];
   max: =x
end (* max *);
. . .
begin end. (* векторы *)
```

Предположим, что код пакета записан в файл *ксн//паскаль//вект*. Использование пакета *векторы:*

```
program пользователь (input, output);
  ¤ пак ксн//паскаль//вект
  var x: real;
     a: array [1 .. 10] of real;
     b1, b2: array [100 .. 1000] of real;
  begin
    (* вычисление значений элементов векторов *)
    x:= max (a);
    writeln ('max (a) = ', x: 10:4);
    writeln ('(b1,.b2) =', скал (b1, b2))
  end. (* пользователь *)
```

Если в программе *пользователь* используется только функция *max*, то прагмат *пак* может иметь вид:

```
¤ пак ксн//паскаль//вект .(max)
```

Пример 2.

Использование пакета стандартных функций на языке Эль-76 в программе на языке Паскаль:

```
program пользователь 1 (output);
  ¤ пак ксн//оспок3//кодфун64
  (* подключен пакет стандартных функций для формата ф64 *)
  var x: real;
  begin
    x := expa (2.0,2);
    (* использована процедура возведения в целую степень *)
  writeln ('2. 0 * * 2 = ',x: 5:1)
  end. (* пользователь 1 *)
```

Пример 3.

Использование пакета на языке Паскаль в программе на языке Эль-76.

```
(*текст пакета на языке Паскаль *)
¤  уст пак
program пакет;
const a = 10;
type t = array [1. . 10] of real;
procedure p (var x: integer);
begin x := x + a end (* p *);
procedure q (var y: integer);
var b: integer;
begin b := 10;
        y := y + b
end (* q *);
begin end. (* пакет *)
```

Текст использующей программы на Эль-76 (*ксн*//*паскаль*//*пакеткод* – внешнее имя кода пакета):

```
начало
    конст n = ксн//паскаль//пакеткод,
          p = прогр (элспр (генув (), n, "p")),
          q = прогр (элспр (генув (), n, "q")),
          печ = читатрзадачи (виртвыв);
    ф64 a := 0, в := 0;
    запф (печ, :"текст на обращение к пакету на Паскале" l);
    запф (печ, a: "начальные значения: a=" g (–3),
               в: "в=" g (–3),: l);
    запф (печ, : "вызов процедуры p" l);
    p (имя a);
    запф (печ, :"вызов процедуры q" l);
    p (имя b);
    запф (печ, a: "конечные значения: a=" g (–3), в: "в=" g(–3), : l)
    конец
```

В результате использования этой программы будет выведен следующий текст:

```
тест на обращение к пакету на Паскале
начальные значения: а= 0 в= 0
вызов процедуры p
вызов процедуры q
конечные значения: а= 10  в = 10
```

2.4.7. Комплексация и пошаговая разработка программ. Многоязыковый комплексатор [25] – общесистемная компонента системы программирования МВК "Эльбрус", предназначенная для сборки программ из процедур, написанных на одном или нескольких языках программирования. Комплексатор, в частности, может применяться для сборки паскаль-программы и для подключения к ней процедуры на другом языке (например, на Эль-76). Управление комплексатором осуществляется на командном языке, реализованном на основе средств определяемого синтаксиса языка Эль-76 (п. 1.32).

Пример 1.

Комплексация паскаль-процедуры и паскаль-программы.

```
(*исходный текст программы *)
program списки (output);
  type список = ↑ элем;
       элем = record инф: integer; след: список end;
  var c: список; p: список;
  function поиск (x: integer): список;
(*поиск элемента x в глобальном списке c *)
begin writeln (' поиск: заглушка' );
        поиск: = nil
end; (*поиск *)
begin
  new (c); c ↑. инф := 1;
  new (p); p ↑. инф: = 2; p ↑. след := nil;
```

```
c ↑. след := p;
p := поиск (1);
if p = nil
then writeln ('не найден' )
else writeln ('найден элемент =' , p ↑. инф)
end. (* ксн//паскаль//кодспис – код программы "списки" *)
```

В исходном варианте программа выведет текст:
поиск: заглушка
не найден

В тексте программы *списки* для функции *поиск* введена заглушка, которая используется при нисходящей разработке и отладке программы. Пусть функция *поиск* с заданным заголовком независимо разработана и отлажена в контексте программы-драйвера:

```
program драйвер (output);
  type список = ↑ элем;
       элем = record инф: integer; след: список end;
  var с: список;
    function поиск (x: integer): список;
      var r: список; b: boolean;
      begin r := c; b := true;
      while (r < > nil) and b
      do
        if r ↑ . инф = x then b := false
        else r := r ↑. след;
        if в then поиск := nil
        else поиск := r
      end (* поиск *);
    begin . . . end. (* драйвер *)
  (* ксн//паскаль//кодотл – код программы-драйвера *)
```

После завершения отладки требуется заменить в объектном коде исходной программы код заглушки функции *поиск* на код ее настоящего тела. Эту задачу решает многоязыковый комплексатор:

```
контекст конст к # глобарх//инткомпл = (глобарх//реалкомпл); к
начало
    арх ксн//паскаль;
    баз //кодспис;
    рез //резкод;
    зам поиск на //кодотл/поиск;
    компл
конец
```

Комплексатор организован как модульный объект, в интерфейс которого входят его директивы. Для использования командного языка комплексатора производится настройка на его контекст. Директива *арх* устанавливает справочник-основу внешних имен. Директивы *баз* и *рез* указывают имена файлов исходного кода и кода-результата комплексации. Директива *зам* обеспечивает замену процедуры *поиск* базовой программы на процедуру *поиск* из указанного файла объектного кода. При замене глобальные

описания исходной процедуры и процедуры-заменителя (в данном случае – переменная *c*) отождествляются по *идентификаторам.* Если они имеют разные идентификаторы (например, в программе *драйвер* глобальная переменная имеет идентификатор *s*), то отождествление можно задать явно (после директивы *зам*):

экв c = s;

В модифицированном варианте программа *списки* выведет текст:

найден элемент = 1

Таким образом, комплексатор удобно использовать не только для обычной сборки программы, но и для ее пошаговой разработки. При комплексации необходимо, чтобы уровень вложенности исходной процедуры и процедуры-заменителя был один и тот же.

Пример 2.

Комплексация паскаль-программы с процедурой на Эль-76.

```
(* исходный текст программы *)
program файлы (intfile, output);
type fi = file of integer; var intfile: fi;
function seek (var f: fi; n: integer): integer;
(* выдает значение n-й компоненты файла f *)
begin writeln ('seek: заглушка');
      seek := maxint
end ; (* seek *)
begin
  reset (intfile);
  writeln ('файл [10] =', seek (intfile, 10))
end. (* объектный код: ксн//паскаль//кодфайл *)
```

В программе требуется описать функцию *seek*, выдающую значение *n*-й компоненты заданного файла целых чисел *f* либо значение *maxint*, если этой компоненты нет (*n* считается от нуля). На языке Паскаль такую функцию невозможно запрограммировать эффективно, так как в нем отсутствует произвольный доступ к файлу. Однако ее легко запрограммировать на Эль-76 и затем воспользоваться комплексатором для замены ее объектного кода:

```
% текст программы на Эль-76 %
начало
  процедура компонента = функция (ф, ном)
  % чтение из файла ф содержимого слова с номером ном. %
  % файл ф предполагается дисковым или барабанным     %
    до * ситлкф
      (ф64 п := позн (ф, локал: ложь),
           длинблок := читатр (ф, максдлинблока),
           рез := чит (п, ном/: длинблок)
                         [ном остат длинблок];
      рез )
```

```
    при ситлкф: цел64 4 "7FFFFFFFFFFFFFFF" %
    всесит ;
    0
конец % ксн//паскаль//кодкомп – код программы
```

Задание комплексатору:

```
контекст конст к # глобарх//инткомпл = (глобарх//реалкомпл) ; к
начало
    арх ксн//паскаль;
    баз //кодфайл;
    рез //резкод;
    зам seek на //кодкомп/компонента;
    компл
конец
```

При комплексации с динамическим языком следует учитывать, что комплексатор не может полностью выполнить контроль соответствия типов, который осуществляется при комплексации Паскаль-Паскаль. Для формальных параметров процедур *seek* и *компонента* он контролирует только совпадение форматов, остальное возлагается на программиста. В частности, ту же процедуру *компонента* можно использовать и для чтения *n*-й компоненты файла вещественных чисел (*file* of *real*).

С точки зрения пользователя системы Паскаль-Эльбрус, комплексатор и аппарат пакетов (п. 2.4.6) решают сходные задачи. Комплексатор более удобно использовать в процессе пошаговой разработки программы и автономной отладки ее процедур; прагмат вызова пакета служит наиболее лаконичным средством для использования в паскаль-программе уже отлаженного пакета процедур.

2.4.8. Диагностика ошибок. При обнаружении синтаксической или семантической ошибки в исходном тексте программы транслятор Паскаль-Эльбрус выдает сообщение в следующей форме:

```
* * * номош * * * ошибка: номстр
        ошибочная строка
                 #
        сообщение об ошибке
```

где *номош* – порядковый номер ошибки, *номстр* – номер ошибочной строки. Под ошибочной строкой в месте ошибки помещается символ #. Например:

```
* * * 5 * * * ошибка: 00007001
  x := y + 1;
  #
* x * – неописанный идентификатор
```

Всего паскаль-транслятор выдает более 250 сообщений об ошибках.

В некоторых случаях с целью нейтрализации ошибки транслятор пропускает часть исходного текста. Например, для языка Паскаль, имеющего унифицированный синтаксис всех классов описаний (типов, переменных, параметров процедур и т.п.), весьма трудно диагностируемой является

следующая ошибка:

```
var z record re, im: real;
    i: integer;
 procedure p; ...
```

Пропущен символ **end** в обозначении типа "запись", но установить место его вставки невозможно, так как описания полей записи по форме не отличаются от описаний переменных. В таких случаях транслятор выдает сообщение о пропуске фрагмента текста, отмечая начало и конец фрагмента, и выполняет пропуск до ближайшего служебного слова, указывающего на начало нового раздела описаний (в данном случае – **procedure**).

Если в исходном тексте паскаль-программы были обнаружены ошибки, то объектный код этой программы запускать не рекомендуется. Однако в некоторых случаях даже при наличии ошибок будет получен правильный объектный код (например, при пропуске символа ";" после описания). Поэтому транслятор не принимает каких-либо мер, предотвращающих запуск ошибочной программы, а оставляет анализ серьезности ошибок на усмотрение пользователя. Некоторые нарушения лексики и синтаксиса не считаются ошибками; они диагностируются предупреждающими сообщениями и исправляются транслятором. Например, если мантисса вещественного числа содержит более 34 цифр, то ее лишние цифры в конце игнорируются.

К прекращению трансляции приводит лишь небольшое число ошибок: неверные параметры транслятора, нарушение возрастающей нумерации строк входного файла (п. 2.4.3), неожиданный конец программы, превышение лимита ошибок *максош* (п. 2.4.2), а также превышение различных ограничений (например, переполнение таблицы). При прекращении трансляции файлу объектного кода присваивается значение атрибута *типфайла* = = *данные*, что предохраняет его от запуска.

2.5. Исполнение и отладка паскаль-программ

2.5.1. Параметры и запуск программы. Программа на языке Паскаль по форме описания и передачи параметров очень близка программе-процедуре языка Эль-76 (п. 1.18.2). Идентификаторы параметров задаются в заголовке программы:

```
program p (input, output, f, g, x)
```

и доопределяются в разделе описаний переменных:

```
var f: file of integer;
    g: file of real;
    x: real
```

Идентификаторы *input* и *output* обозначают стандартные текстовые файлы ввода-вывода и не требуют доопределения. Все параметры программы являются параметрами-переменными и, как правило, файлами, но допустимы и другие типы параметров.

При запуске программы число, типы и порядок фактических параметров должны соответствовать описанию формальных параметров в тексте паскаль-программы, иначе фиксируется ошибка. Ошибки, связанные с

тем, что фактическим параметром является внешний или оперативный объект, отличный от файла, обнаруживаются не в момент передачи, а при выполнении первой операции над файлом (*reset* или *rewrite*).

Фактические параметры, соответствующие формальным параметрам типа *text* (в частности, стандартным файлам *input* и *output*) должны быть текстовыми файлами на диске, барабане, терминале или АЦПУ. Фактические параметры, соответствующие формальным параметрам типа **file of** *t* или **packed file of** *t*, где *t* – некоторый тип, должны быть *файлами данных* на диске или барабане (т.е. значение атрибута *типфайла* должно быть равно нулю). Остальные фактические параметры должны быть *переменными*. Допускаются также *параметры-значения*, например, числа (при этом, в отличие от многих систем программирования для традиционных ЭВМ, значения чисел не портятся при присваивании параметру). Динамический контроль типов и значений осуществляется аппаратно, при выполнении операций.

В *диалоговом режиме* паскаль-программа запускается на исполнение директивой *исп* (п.1.33.10):

*исп //ркод (//ввод, э, //дан*1, *//дан*2, 1.0),

где *//ркод* – файл объектного кода программы *р*, *//ввод* – текстовый файл, *//дан*1 и *//дан*2 – файлы данных. Вывод осуществляется на терминал. Пятый параметр – значение типа *real*.

В *пакетном режиме* для запуска паскаль-программы необходимо ее открытие (п.1.30.1), после чего программа запускается как процедура с заданным набором фактических параметров:

```
начало
    конст к = ксн//паскаль;
    ф64 а := 3.14;
    прогр (к//ркод) (к//ввод, читатрзадачи (виртвыв),
                     к//дан1, к//дан2, имя а)
конец
```

В этом примере вывод производится в файл виртуального вывода задачи, а в качестве пятого параметра передано имя переменной со значением типа *real*.

2.5.2. Особенности ввода-вывода. Ввод-вывод при исполнении паскаль-программы осуществляется средствами системы простых файлов и форматного обмена. Операции *reset* (*f*) (установка файла для чтения) и *rewrite* (*f*) (установка файла для записи) соответствует создание позиционной переменной. При этом операция *rewrite* создает и сам файл *f*, если он является локальным в программе (не передан в качестве параметра):

var *f*: **file of** *real*

Операции чтения и записи *read, write, readln, writeln* для текстовых файлов реализованы с использованием процедур форматного обмена. Операциям над файлами данных (*get, put, read, write*) соответствуют в реализации операции буферизованного обмена с позиционной переменной.

Во всех операциях обмена осуществляется динамический контроль параметров. Например, если в операцию *get*(*f*) передан еще не существующий файл или файл, над которым не выполнена операция *reset*(*f*), то исполнение программы прекращается и выдается сообщение об ошибке.

При завершении программы все внешние файлы, переданные через параметры, закрываются системой Паскаль-Эльбрус (напомним, что специальной операции закрытия файла в языке Паскаль не предусмотрено). Локальные файлы, все ссылки на которые сосредоточены только в паскаль-программе, создаются на барабане и уничтожаются автоматически по завершении задачи средствами системы простых файлов МВК "Эльбрус". Каких-либо средств сохранения созданных локальных файлов в системе Паскаль-Эльбрус нет. Таким образом, обмен файлами между различными паскаль-программами должен осуществляться через параметры. При этом гарантируется корректная обработка файла, если тип файла в программе, читающей файл, структурно идентичен типу файла в программе, его формирующей.

Пример.

```
program создатель (f);
var f: file of real; i: integer;
begin
   rewrite (f);
   for i := 1 to 1000 do
     write (f, 1/i)
end . (* //создкод – объектный код программы *)
program пользователь (g, output);
var g : file of real; r: real;
begin
   reset (g);
   while not eof (g) do
   begin
      real(g, r ) ; write (r)
   end
end . (* // пользкод – объектный код программы *)
```

В результате запуска программы *создатель*:

исп//создкод (//данные)

будет создан файл данных (*типфайла* = 0) из 1000 вещественных чисел, обратных целым числам от 1 до 1000. При исполнении программы *пользователь*:

исп//пользкод (//данные, э)

эти числа будут выведены на экран.

Терминальный ввод-вывод. В системе Паскаль-Эльбрус возможен ввод текстовых данных с терминала и вывод их на терминал, если соответствующий формальный параметр имеет тип *text*.

Для *ввода* (в операциях *read* и *readln*) выдается стандартное приглашение ">". Операции чтения *readln* соответствует конец сообщения; данные для предшествующих операций *read* можно набрать в том же сообщении (при недостаточности числа исходных данных будет выдано дополнительное приглашение). Для указания конца входного терминального файла следует поместить курсор под символом приглашения >, набрать на его месте символ @ и нажать клавишу "ввод".

При *выводе* на терминал операция *writeln* соответствует переходу на новую строчку текста на терминале. Выходные данные выдаются порциями; при исчерпании экрана выдается приглашение "?", на которое для продолжения вывода следует ответить пустой посылкой.

Пример.

Диалоговая программа-калькулятор:

```
program калькулятор (input, output);
(* программа – калькулятор *)
 var a, b: integer; op: char;
 begin writeln ('* * * калькулятор * * *');
 writeln (' программа выполняет операции над целыми числами ');
 writeln (' исходные данные вводятся в виде: m op n');
 writeln (' где m и n – целые числа, op – операция +, –, *, / ');
 while not eof do
     begin
         writeln (' введите исходные данные: ');
         readln (a, op, b);
         if op in [' + ', '–', '*', '/']
         then
         case op of
         '+': writeln ('результат =', a + b);
         '–': writeln ('результат =', a – b);
         '*': writeln ('результат =', a*b);
         '/': writeln (' результат =', a div b)
         end
         else writeln (' неверный знак операции!')
     end ; (* while *)
     writeln ('* * * конец работы калькулятора * * *')
 end . (* // калькулятор – объектный код программы *)
```

Пример.

Сеанс диалога с программой *калькулятор*:

```
исп // калькулятор (э, э)
* * * калькулятор * * *
программа выполняет операции над целыми числами
исходные данные вводятся в виде: m op n,
где m и n – целые числа, op – операция +, –, *, /
введите исходные данные:
>1 + 2
результат = 3
введите исходные данные:
>5 : 10
неверный знак операции!
введите исходные данные:
@
* * * конец работы калькулятора * * *
```

2.5.3. Представление данных. Для работы с отключенным контролем типов и для корректного использования в паскаль-программе процедур на других языках программисту необходимо знать особенности представле-

ния данных. Способ представления значения константы или переменной определяется ее типом. Поскольку сложный тип может быть упакованным (п. 2.3.4), для каждого типа указаны два варианта представления – обычное и упакованное. Последнее применяется, если значение указанного типа является непосредственной компонентой упакованного типа (массива, записи или буферной переменной файла).

Стандартные типы:

- *integer* – целое 64 (упакованное – 64 бита);
- *real* – вещественное 64 (упакованное – 64 бита);
- *boolean* – набор длины 1 (упакованное – 1 бит);
- *char* – набор длины 8 (упакованное – 1 байт);
- *shortint* – целое 32 (упакованное – 32 бита);
- *shortreal* – вещественное 32 (упакованное – 32 бита);
- *longreal* – вещественное 128 (упакованное – 128 битов);
- *alfa* – набор длины, кратной 8 (упакованное – 8 байтов) .

В упакованном представлении целые и вещественные числа хранятся без тегов (т.е. соответствующие слова имеют тег "набор").

Типы-перечисления $(c_0, c_1, \ldots, c_{n-1})$. Неупакованное представление – целое 64 (номер элемента). Упакованное представление – минимально необходимое число битов: **перв**1 $(n - 1) + 1$ битов (n – число элементов).

Типы-диапазоны $l \,..\, u$. Неупакованное представление совпадает с представлением l или u. Упакованное представление занимает минимальное число битов: если l или u – отрицательные целые числа, то **перв**1 $\max(|l|, |u|) + 2$ битов (с учетом знака), иначе – **перв**1 $ord(u) + 1$ битов, где **перв**1 – номер первой единицы в битовом представлении числа (п. 1.4).

Сложные типы. Для значений сложного типа принято линейное представление: так как размер любого значения известен статически, значения компонент выстраиваются в общей области памяти. Значение сложного типа представлено как *вектор* некоторого *формата* и *длины*, возможно, являющийся подмассивом другого вектора. Например, значение переменной

var *a*: **array** [1 .. 10] **of record** *re*, *im*: *real* **end**

представляется вектором из 20 слов, указатель которого хранится в области локальных данных.

Если сложный тип является упакованным (имеет признак **packed**), то для его непосредственных компонент применяется упакованное представление. Например, значение переменной

var *b*: **packed array** [1 .. 5] **of** (*один, два, три*)

представляется набором из 10 битов, а переменной

var *c* : **packed array** [1 .. 50] **of** *char*

– строкой из 50 байтов.

Массивы **array** $[l_1 \,..\, u_1, \ldots, l_n \,..\, u_n]$ **of** t размещаются *по строкам.* Паспорт массива, как правило, во время исполнения не строится, так как границы l_i, u_i известны статически. Если массив передается параметром в процедуру, формальный параметр которой специфицирован как *схема формальных массивов* (п.2.3), то для него строится *паспорт* стандартной струк-

туры, содержащий шаги по измерениям, компенсирующий член, указатель базового вектора, верхние и нижние границы. Длина паспорта равна $(n + 3)/: 3 + 1 + (2n + 2)/: 3$ слов, где n – число измерений (значение каждого слагаемого равно длине соответствующей компоненты паспорта).

Если обращение к многомерному массиву выполняется в цикле, причем индексы являются *линейными формами* от параметров цикла:

```
for i := 1 to n do
  for j := 1 to m do
  a[i * 2 + 1, j * 3] := b[i, j] ,
```

то с целью оптимизации компилятор строит и размещает в области локальных данных паспорт массива в сокращенном виде (без значений границ), длиной $(n + 3)/: 3 + 1$ слов (n – число измерений), по которому выполняется аппаратная операция обращения к элементу массива.

Записи: **record** $f_1 : t_1 ; \ldots ; f_n : t_n$ **end** представляются последовательностью значений полей f_i в их текстуальном порядке. Формат области памяти, занимаемой записью, равен максимальному из форматов компонент, длина выравнена по этому формату. Переменная-запись размещается в области локальных данных, если она не упакована, не содержит компонент-массивов и больших множеств. Такая запись называется прямоадресуемой. Прямоадресуемая запись рассматривается как совокупность переменных-полей. Например:

```
var r : record n : alfa; a : 0 . . 200; s : (m, f) end.
```

Значение переменной r занимает три слова в области локальных данных: по одному слову – значения полей $r.n$, $r.a$ и $r.s$. Запись, не являющаяся прямоадресуемой, представляется, как и массив, вектором в математической памяти.

Множества **set of** t. Неупакованное представление зависит от типа t. Если t – тип "целое", "перечисление", "логическое" или диапазон одного из этих типов, то множество приводится к типу **set of** $l\, .\, .\, u$ ($ord(l) = 0$, $ord(u) = 63$) и представляется набором из n битов (n – число элементов типа t), размещаемом в слове (малое множество). Если же t – тип или диапазон типа "символ" (*char*), то значение представляется четырьмя словами. Упакованное представление множества (**packed set of** t) занимает минимальное число битов, равное числу элементов типа t.

Неупакованные значения с упакованными компонентами. Если значение переменной имеет неупакованный сложный тип, а тип компоненты имеет признак **packed**, то компонента размещается как упакованная, но с выравниванием до слова. Такое представление позволяет сочетать более быстрое обращение к массиву слов для выбора компоненты с экономией памяти внутри компоненты.

Пример.

Описание таблицы идентификаторов

```
const длтаб = 1000;
var таб: array [1 . . длтаб] of
         packed record ид: alfa; длина: 0 . . 8;
                  ссылка1, ссылка2 : 0 . .длтаб end;
```

Представление таблицы будет занимать 2000 слов, каждого элемента – два

слова, причем обеспечивается оптимальный доступ к полям элемента (с помощью операций выделения и вставки поля, п. 1.4.3).

Файлы. Файл представляется как последовательность компонент во внешней памяти. Каждая компонента размещена по тем же правилам, что и компоненты других структурированных типов. Переменная-файл *f* представлена позиционной переменной, ссылающейся на файл. В позиционной переменной, в частности, хранится значение буферной переменной *f*↑ – текущая компонента файла.

Указатели. Каждому обозначению типа "указатель" ↑*t* ставится в соответствие *пул* – вектор переменной длины, в котором размещаются динамические переменные типа *t*. Такое представление позволяет распределять память для динамических переменных типа *t* независимо от других типов. Указатель пула хранится в области локальных данных той процедуры, в которой введено обозначение типа ↑*t*. Значение-указатель типа ↑*t* представляется *индексом* в пуле. Следовательно, его упакованное представление имеет длину 20 битов.

2.5.4. Отладка программ. В заключение главы приведем необходимые сведения для отладки паскаль-программ: тексты и смысл сообщений о динамических ошибках, форму символьной распечатки стека, некоторые способы символьной отладки программ.

Сообщения о динамических ошибках. При динамической ошибке в паскаль-программе выдается подробное сообщение и символьная распечатка стека. Некоторые динамические ошибки (например, переполнение в целочисленной арифметической операции) обнаруживаются аппаратно. Ниже приведены тексты сообщений о динамических ошибках, которые обнаруживаются либо командами динамического контроля, генерируемыми паскаль-транслятором, либо процедурами системы динамической поддержки системы программирования Паскаль-Эльбрус. В необходимых случаях пояснен смысл сообщений и даны рекомендации по исправлению ошибок. Тексты сообщений выделены курсивом.

1. **** файл @ *** файл не открыт.*

В операции "буфер файла" *f*↑ над файлом *f* не выполнена операция *reset.*

2. **** файл @ *** в операцию подан не файл или файл не открыт.*

В операции "буфер файла" *f*↑ над файлом *f* не выполнена операция *rewrite.*

3. **** reset *** для ввода открывается строго выводной файл.*

Такое сообщение может быть выдано, например, в случае, если операция *reset*(*f*) выполняется над файлом-параметром *f* таким, что фактический параметр является АЦПУ-файлом.

4. **** rewrite *** для вывода открывается строго вводной файл.*

Пример.

Программе передан фактическим параметром файл для чтения с перфокарт, над которым выполняется операция *rewrite.*

5. **** reset *** в операцию подан не файл.*

Динамический контроль типов для внешнего файла.

6. **** rewrite *** в операцию подан не файл.*

Смысл аналогичен сообщению 5).

7. **** чтение *** в операцию подан не файл или файл не открыт.*

Аргумент операции *get, read* или *readln* — не файл либо файл, над которым не выполнена операция *reset.*

8. * * * *запись* * * * *превышен максимальный размер файла.*

В операции *put, write* или *writeln* превышено значение атрибута *максдлинфайла.* Рекомендуется увеличить значение атрибута:

ат//выхфайл мдф: 20

или проверить задачу на зацикливание.

9. * * * *чтение* * * * *ошибка в изображении числа.*

В операции *read* или *readln* над текстовым файлом в исходном файле обнаружено неверное изображение числа.

10. * * * *чтение* * * * *попытка чтения за концом файла.*

Попытка выполнения операции *get, read* или *readln* над файлом *f*, если *eof*(*f*) истинно.

11. * * * *запись* * * * *в операцию подан не файл или файл не открыт.*

Аргумент операции *put, write* или *writeln* — не файл либо файл, над которым не выполнена операция *rewrite.*

12. * * * *чтение* * * * *файл открыт для записи.*

В операцию *get, real* или *readln* подан файл *f*, над которым не выполнена операция *reset*, но выполнена операция *rewrite.*

13. * * * *eof* * * * *в операцию подан не файл или файл не открыт.*

Смысл аналогичен сообщениям 7) и 11).

14. * * * *eof* * * * *файл не открыт.*

В операцию *eof*(*f*) подан файл *f*, над которым не выполнена операция *reset* или *rewrite.*

15. * * * *eoln* * * * *в операцию подан не файл или файл не открыт.*

Смысл аналогичен сообщениям 7), 11) и 13).

16. * * * *запись* * * * *файл не открыт для записи.*

В операцию *put, write* или *writeln* подан файл, над которым не выполнена операция *rewrite.*

17. *Число фактических параметров программы не равно числу формальных.*

Динамический контроль параметров программы.

18. Выход за границу типа-диапазона при присваивании

Пример.

var *x* : 1 .. 10; *y* : *integer*;
y := 12; *x* := *y*

Ошибка возникает при втором присваивании.

19. *Выход за границу базового типа множества при присваивании.*

Пример.

var *s*: **set of** 1 .. 5;
s := [0, 1]

20. *Второй аргумент в операции* * *mod* * < 0.

Согласно семантике стандартного языка Паскаль в операции *x* **mod** *y* ошибкой считается, если $y < 0$.

21. *Выход за границу массива при индексации.*

Система Паскаль-Эльбрус контролирует программным путем выход за границу массива по каждому измерению. При отключении динамичес-

ких проверок (¤ *уст0 дин*) выполняется только аппаратный контроль выхода за границу вектора.

22. *Недопустимый аргумент в операции * chr *.*

В языке Паскаль-Эльбрус допустимым считается аргумент x операции $chr(x)$, если $x >= 0$ и $x <= 255$.

23. *Недопустимый аргумент в операции * pred *.*

В операции $pred(c)$ значение c – начальное значение типа "перечисление", либо выполняется операция $pred(-maxint)$.

24. *Недопустимый аргумент в операции * succ *.*

В операции $succ(c)$ значение c – конечное значение типа "перечисление", либо выполняется операция $succ(maxint)$.

25. *Переполнение в операции с целым аргументом.*

Значение выражения, содержащего операцию $sqr(i)$ (где i – целое число типа *integer* или *shortint*), превышает максимальное целое значение этого типа.

26. *В операторе * case * нет альтернативы с заданным значением.*

Пример.

```
x := 0;
case x of
1: writeln ('один');
2: writeln ('два');
3: writeln ('три')
end
```

27. *Ограничение реализации: размер пула превысил* 2^{20} *слов.*

Переполнение памяти для динамических переменных при выполнении процедуры *new*.

28. *Число признаков в * dispose * не равно числу признаков в * new *.*

Динамическая переменная, созданная вызовом процедуры $new(p, c_1, \ldots, c_n)$, освобождается вызовом процедуры $dispose(p, d_1, \ldots, d_m)$, где $m \neq n$.

29. *Значение признака в * dispose * не равно значению признака в * new *.*

Динамическая переменная, созданная вызовом процедуры $new(p, c_1, \ldots, c_n)$, освобождается вызовом процедуры $dispose(p, d_1, \ldots, d_n)$, причем существует k такое, что $c_k \neq d_k$.

30. *В данном вызове * dispose * нельзя передавать значения признаков.*

Динамическая переменная, созданная вызовом процедуры $new(p)$, освобождается вызовом процедуры $dispose(p, c_1, \ldots, c_n)$.

После выдачи сообщения об ошибке система динамической диагностики системы программирования Паскаль-Эльбрус выполняет структурный переход по стандартной ситуации, соответствующей смыслу ошибки: *неверныйоперанд, границамассива, недопустимыйаргумент* или *переполнцел64.* Возникновение стандартной ситуации приводит к выдаче символьной распечатки стека и прекращению паскаль-программы.

Форма символьной распечатки стека. Символьная распечатка стека выдается системой динамической диагностики ОСПО МВК "Эльбрус" по состоянию стека и расширенному файлу объектного кода паскаль-программы. При запуске программы в диалоге она выдается

на терминал, в пакете – в файл виртуального вывода. Способ управления системой динамической диагностики в диалоге описан в п. 1.33.10.

Символьная распечатка стека (в режиме *печ ч*) содержит следующую информацию:

– сообщение о динамической ошибке;

– номер ошибочной строки в исходном тексте;

– фрагмент исходного текста, содержащий ошибочную строку;

– операнды текущего выражения, вычисление которого на завершено (значения операндов, вычисленные последними, выдаются первыми);

– последовательность вызванных процедур паскаль-программы, в порядке, обратном их вызову. Для каждой процедуры выдается ее имя, диапазон строк, номер строки вызова следующей процедуры; если системе задан ключ "все" – имена и значения локальных переменных. Для текущей прерванной процедуры имена и значения локальных переменных выдаются всегда. Локальные данные выдаются в алфавитном порядке их имен, в форме: *идентификатор = значение.*

Форма выдачи локальных данных определяется их типом и представлением данных (п. 2.5.3). Числа выдаются в форме, близкой к их изображениям; наборы – в шестнадцатеричном виде с указанием их длины: *наб8 C1* обозначает набор "A". Неопределенные значения простых переменных выдаются в виде пустых объектов: *пусто64, пусто32.* Переменные сложных типов, представленные указателями векторов, выводятся в виде

деск64 [10] ⟨20107700⟩ .

где 64 – формат вектора, 10 – длина, в угловых скобках – адрес. Поля прямоадресуемой переменной-записи *r* (п. 2.5.3) выводятся с идентификаторами *r*.1, *r*.2 и т.д. Идентификаторы *л.локдок*.1 и *л.локдок*.2 обозначают два слова системной информации – *документацию на локальную память,* используемую ОС для управления локальными массивами. Идентификатором *п.пасдин* обозначен указатель на область интерфейса системы динамической диагностики системы программирования Паскаль-Эльбрус. Значения переменных-файлов выдаются в виде: *позп* (*терм*) ⟨20001000⟩ (позиционная переменная на терминальный файл).

Пример.

Символьная распечатка стека:
ошибка в задаче

```
# # # ######################
# # # причина: неверный операнд     ###
# # # место: p. 00004000            ###
# # # ######################
00000000 program ошибка (output);
00001000 var x: real;
00002000 procedure p(a: real);
00003000 begin
** 00004000 ****** x := x + a *************
00005000 end (* p *);
00006000 begin p(1.0) end.   (* ошибка *)
```

```
    пусто64
**********************************
a = 1.0
************************* p (00002000–00005000)
      x = пусто64  output = позн (терм) ⟨2000C100⟩
********************** ошибка (00000000–00006000)
```

Символьная отладка программ может осуществляться с использованием символьной распечатки стека. Как правило, имеющейся в ней информации вполне достаточно для быстрого обнаружения ошибки. В случаях, когда выполнение программы не прекращается, но необходимо проследить за изменением значений переменных, наиболее удобно воспользоваться директивами *управления процессами* языка стандартного диалога в сочетании с директивой *л* выдачи символьной распечатки стека (п. 1.33.11):

```
> исп//код (э)
  запущен процесс :5
> ост :5
> л :5
```

На экран выдается символьная распечатка стека, в которой содержатся значения переменных. Исполнение программы можно продолжить директивой:

```
> ак :5
```

В случае особо сложных ошибок, связанных с динамикой исполнения программы, помимо традиционных методов отладки (трассировка, контрольные печати) рекомендуется воспользоваться символьным отладчиком ОСПО МВК "Эльбрус".

ГЛАВА 3

ЯЗЫК ПРОГРАММИРОВАНИЯ КЛУ И СИСТЕМА КЛУ-ЭЛЬБРУС

Данная глава имеет двоякую цель. Во-первых, в ней рассматриваются проблемы практического применения концепции абстрактных типов данных (АТД) и дается обоснование выбора языка Клу [83] в качестве возможной основы для использования АТД как технологии практического программирования. Язык Клу недостаточно известен в СССР: на русском языке его описание до сих пор не было опубликовано, несмотря на значительную роль этого языка в развитии концепции АТД и в дальнейшей разработке языков программирования. Данная глава восполняет этот пробел в той мере, в какой позволяют это сделать объем и тематика настоящей публикации. Параграфы 3.1–3.10, посвященные практическим аспектам АТД и языку Клу, носят в некоторой степени самостоятельный характер и связаны с системой "Эльбрус" как описание ядра входного языка системы программирования Клу-Эльбрус.

Во-вторых, заключительная часть главы содержит описание системы Клу-Эльбрус [18] – первой отечественной реализации языка Клу. Система Клу-Эльбрус – языковая основа и программная поддержка для практического применения технологии АТД в системе "Эльбрус". Некоторые вопросы реализации АТД в системе "Эльбрус" рассмотрены в гл. 6.

3.1. Практические аспекты абстрактных типов данных

Принципы АТД, сформулированные в работах Н. Вирта, О.-Й. Дала, Ч. Хоара, Д. Парнаса и Ф. Морриса в конце 60-х – начале 70-х годов, возникли из потребностей практического программирования в повышении модульности, надежности и наглядности программ. Суть концепции АТД состоит в том, что создание и обработка объектов некоторого (абстрактного) типа возможны только через определенный набор операций, связанный с этим типом. Тем самым, программист абстрагируется от конкретного представления объектов данного типа и от реализации операций. Конкретное представление и реализация недоступны (инкапсулированы в определении типа). Важным элементом концепции АТД является аксиоматика, определяющая свойства типа и любой его реализации. Она часто носит неформальный характер и оформляется в виде комментариев, но даже и в

этом случае играет важную роль для повышения наглядности программы. Принцип абстракции данных был теоретически обоснован Ч. Хоаром в работе [76]. С тех пор изучению АТД было посвящено огромное число теоретических работ [2]. Однако практическое применение принципов АТД долгое время (до конца 70-х годов, а в массовом программировании – по настоящее время) было весьма ограничено. Основные причины этого, по-видимому, состоят в следующем.

Отсутствие реализаций языков с АТД. Наиболее ранним предшественником языков с АТД можно считать язык Симула-67 [27], однако в нем нет средств инкапсуляции. Первые версии языков Клу и Альфард [2], в которых концепция АТД отражена в наиболее полном виде, появились в середине 70-х годов, но из них только язык Клу был реализован и использовался для практического программирования [83], а язык Альфард в большей степени был создан как язык накопления правильных алгоритмов и их верификации, а не как язык программирования. В 1979 году был опубликован первый вариант языка Ада, который содержит конструкции, достаточные для выражения концепций АТД (пакет, приватный тип, задача). Однако язык Ада был реализован только в 80-х годах. В 1978 г. разработан язык программирования Модула-2 [15], в котором элементы АТД весьма ограничены (уникальные модули без статических параметров с фиксированными связями в виде описаний экспорта-импорта). За это время появился и ряд других языков с элементами АТД, которые не получили столь широкой известности.

Семантический разрыв между языками программирования и архитектурой ЭВМ [48]. Эта тенденция, неоднократно обсуждавшаяся в научной литературе, сдерживает прежде всего реализацию и использование наиболее современных средств языков программирования, в частности, элементов АТД. Проблемы реализации АТД рассмотрены в работе [69], в которой показано, что эффективная реализация АТД в полном объеме на ЭВМ традиционной архитектуры вызывает большие трудности. Это приводит к наложению существенных ограничений на реализуемый набор элементов АТД (например, запрещению генераторов абстрактных объектов и параметризованных АТД).

Инерция программистов при освоении новых методов программирования. При недостаточности языковых элементов АТД единственным способом использования этих концепций как технологических принципов программирования является их моделирование на доступных языках с помощью простых модульных средств (процедур, макроопределений, библиотечных текстовых вставок). При этом конкретное представление абстрактных объектов, выражаемое совокупностью описаний констант, типов и переменных, остается доступным программисту, так как большинством языков не обеспечивается важнейший элемент АТД – инкапсуляция. Это приводит к тому, что инкапсуляция конкретного представления становится лишь элементом технологической дисциплины, соблюдаемой по соглашению внутри коллектива программистов, которая неизбежно вступает в противоречие с распространенным (во многом верным) мнением о желательности знания конкретного представления всех структур данных и алгоритмов для разработки высококачественных программ. При работе с элементами конкретного представления,

действительно, может несколько повыситься эффективность программы, но значительно снижается ее надежность. Например, если в результате модификации программы элемент данных *a* объекта *x* будет размещаться не во втором слове конкретного представления *x*, а в третьем, то необходимо просмотреть весь текст программы и внести изменения во все обращения к элементу *a* (при этом велика вероятность того, что хотя бы одно обращение не будет замечено и откорректировано). К сожалению, в массовом программировании до сих пор соображениям эффективности (без каких-либо строгих ее оценок) чаще всего отдается предпочтение перед надежностью и наглядностью, хотя применение принципов АТД может обеспечить эти свойства программы без ухудшения ее эффективности. Простое преобразование программы – описание и использование операции вида $get_a(x)$ для обращения к элементу *a* объекта *x* – обеспечивает и защиту от ошибок несанкционированного доступа, и легкость модификации программы, а также сохраняет ее эффективность, если операция get_a реализована как макроопределение или открытая процедура.

Программистов отталкивает кажущаяся сложность терминологии, принятой в языках с АТД, и наличие новых конструкций (итераторов, ситуаций, параметризованных АТД и других), в практической полезности которых они еще не имели возможности убедиться. Однако, как будет видно из дальнейшего содержания этой главы и приводимых в ней примеров программ, все эти конструкции и понятия выражают в несколько более общей форме те методы и приемы программирования, которые уже давно используются на практике.

Элементы современной концепции АТД. Выбор (разработка) и эффективная реализация базового языка АТД являются, как видно из приведенных рассуждений, наилучшим способом внедрения технологии АТД. Сформулируем критерии, которым, по современным представлениям, должен удовлетворять базовый язык АТД (т.е. элементы АТД, которые он должен содержать).

1. *Базовая конструкция для определения АТД* (кластер, форма, капсула и т.п.). Ее составными частями должны быть:

– *имя* (идентификатор) АТД;

– *интерфейс* АТД – совокупность *интерфейсов* его *операций*. Интерфейс каждой операции содержит ее *имя, типы аргументов* и *результатов*, возможные *исключительные ситуации* ее завершения (например, исчерпание стека) и описание *эффекта* ее *выполнения* над абстрактным объектом. Эффект выполнения операции может быть выражен в форме словесного комментария. В интерфейс АТД может быть включена также денотационная семантика (аксиоматика) типа.

Однако, по-видимому, для массового программирования неприемлемо использование методов формальной спецификации программ в их нынешней форме [1] при описании семантики операций: эти методы слишком громоздки, спецификационные конструкции превышают по сложности саму программу и доступны лишь высококвалифицированным специалистам с усиленной математической подготовкой в области алгебры и логики. Разумеется, это не умаляет значения формальных спецификаций в исследованиях по теоретическому программированию;

– *реализация* АТД – совокупность *конкретного представления* абстрактных объектов данного типа и *реализаций* (тел) *операций*, выраженных в терминах конкретного представления.

Набор операций АТД связывается с типом, а не с объектом и, следовательно, не дублируется при создании нового объекта. В этом основное отличие АТД от понятия пакета, введенного впервые в язык Ада и иногда ошибочно отождествляемого с АТД. *Пакет* – это конструкция для обработки конкретной совокупности данных с помощью набора процедур, описанных вместе с этими данными. АТД – совокупность *способов* (прав) *доступа* ко всем объектам заданного абстрактного типа. Число абстрактных объектов не фиксируется, но при обработке каждого из них должны быть указаны *ссылка на тип* (набор операций) и *имя* конкретной *операции*.

2. *Генератор абстрактного объекта* – конструкция для создания экземпляра абстрактного объекта заданного типа. Она может иметь форму одной из операций, входящих в определение АТД (операция *create* в языке Клу), либо универсальной полиморфной операции, входящей в ядро языка (операция new в языке Симула-67). Возможность свободной генерации абстрактных объектов, как отмечалось выше, является преимуществом АТД по сравнению с более статичным понятием пакета.

3. *Параметризованные* (родовые) *АТД* – форма описания АТД для определения не одного типа, а целого класса типов, имеющих аналогичную структуру и отличающихся значениями параметров (констант и типов): например, список с параметром – типом элементов; таблица с параметрами – размером таблицы и типом элементов и т.п. Параметризованные АТД удобно использовать при описании *полиморфных операций*, алгоритмы которых не зависят от параметров-типов либо зависят от них лишь фиксированным образом, требуя от параметра-типа наличия определенных операций. Подстановкой конкретных значений параметров (конкретизацией) из определения параметризованного АТД получаются определения новых АТД заданного класса, например: список целых чисел; таблица анкет размером 1000.

Возможность параметризации пакетов (в описанном выше смысле) есть в языке Ада. Язык Модула-2 не допускает параметризации модулей: каждый из них существует в неизменном и единственном экземпляре.

4. *Абстрактные исключительные ситуации* – средства обработки ошибок в абстрактных операциях. Каждая операция кроме нормального исхода работы (выдачи результата, изменения состояния объекта) может иметь один или несколько *исключительных исходов*, каждый из которых связан с возникновением определенной *ситуации* (выхода за границу массива, отсутствия элемента в списке и т.п.). Практика программирования (например, опыт использования языка Эль-76) показывает, что механизм ситуаций является наиболее удобным способом обработки исключительных исходов операций (п.1.15). К решению этой проблемы имеются и альтернативные подходы. В языке *VAL* [79] введены специальные *ошибочные значения* (*error values*), выдаваемые в качестве результата при исключительных исходах операций. Такой подход более естествен в теоретических работах (доопределение функции), но при программировании он неудобен, так как либо требует явных проверок на ошибочные значения в каждой

операции (для локализации ошибки), либо приводит к неконтролируемому распространению ошибочных значений по всей программе.

Механизм ситуаций включен в языки Эль-76 и Ада.

5. *Итераторы* (абстракции управления) – модули для управления поэлементным перебором компонент абстрактных объектов сложной структуры (массивов, списков, графов, множеств и т.п.). Как правило, в практике программирования поэлементный перебор осуществляется с помощью циклов или рекурсивных процедур. При этом один шаг перебора можно разделить на два независимых этапа: *получение* компоненты и ее *обработку*. При переходе к следующей компоненте используется информация о конкретном представлении структурированного объекта. Поэтому наиболее технологичным решением представляется локализация способа перебора в специальном модуле – *итераторе*, который используется совместно с циклом обработки компонент объекта. Итераторы введены в языки Клу и Альфард. В остальных языках (в том числе Ада и Модула-2) итераторов нет, но ввиду практической потребности в подобных конструкциях они моделируются с помощью процедур с процедурными параметрами. Например, рекурсивная процедура *обход* (t, p), где t – ссылка на двоичное дерево, p – процедура, задающая действия при обработке вершины, по существу, является итератором, в котором скрыт способ обхода дерева, а вызов процедуры p моделирует исполнение тела цикла обработки вершин дерева.

Кроме перечисленных компонент, относящихся собственно к АТД, базовый язык АТД должен иметь комплект *стандартных типов* (целое, вещественное, логическое, строка, символ и т.д.) и *стандартных конструкторов типов* (массив, запись, последовательность, объединение и т.п.), достаточный для конкретного представления сложных абстрактных объектов. Стандартные конструкторы типов могут быть введены, например, как параметризованные АТД.

Другие требования к базовому языку АТД. Новая технология программирования должна быть удобна для программиста и не должна вызывать дополнительных трудностей, связанных с изучением громоздкого языка и использованием его сложных конструкций. Кроме того, применение нового языка маловероятно, если его реализация существенно уступает по эффективности реализации более привычных языков, даже при всех преимуществах нового языка. Из этих соображений вытекают следующие естественные требования к базовому языку АТД:

– обозримость языка, ортогональность конструкций, концептуальная эффективность (лаконичность). Язык должен быть построен на нескольких фундаментальных понятиях и не должен содержать "лишние" конструкции;

– простота изучения и использования; программы на базовом языке должны быть ясными и наглядными, а семантика – достаточно простой; язык должен быть концептуально богаче своих предшественников, но структурно проще;

– возможность эффективной реализации.

Уместно провести аналогию с языком Лисп, который до настоящего времени является базовым языком большинства разработок по искус-

ственному интеллекту благодаря своей ясности, концептуальной экономности и наличию операций, удобных для этих задач. Базовый язык АТД должен обладать близкими качествами и играть аналогичную роль в технологии производственного программирования.

Выбор базового языка АТД. Из рассмотренных языков, наиболее известных и популярных в настоящее время (Ада, Модула-2), и из менее известных языков с АТД (Клу, Альфард, Mesa, Euclid, Asbal, EL/1 [2]) сформулированным требованиям в наибольшей степени удовлетворяет язык Клу. Он, с одной стороны, достаточно прост по семантике и синтаксису, с другой стороны, содержит все перечисленные элементы АТД в полном объеме. Как показал опыт разработки системы программирования Клу-Эльбрус [18], язык Клу допускает реализацию с приемлемой эффективностью в достаточно короткий срок, хотя он и сложнее в реализации, чем более традиционные языки (например, Паскаль). Язык Клу пока не приобрел столь широкого распространения, как Ада или Модула-2, но его элементы использованы при разработке других языков с АТД: Декарт [7], Атлант [33]. В сравнении с языком Клу язык Ада значительно более громоздок, не обладает свойством ортогональности (например, в языке Ада можно смоделировать АТД тремя способами: пакетом, приватным типом, задачей) и не содержит некоторых необходимых элементов АТД (генераторов абстрактных объектов, итераторов). Язык Модула-2, как уже отмечалось, имеет весьма ограниченные возможности абстрактных данных. Кроме того, цели создания языков Ада и Модула-2 совершенно иные, нежели языка Клу; он создавался именно как базовый язык программирования с АТД.

3.2. Обзор основных понятий языка Клу

Язык Клу (Clu) [83] создан коллективом специалистов Массачусетского технологического института (США) под руководством проф. Б. Лисков. Новый вариант языка Клу, который описан в этой главе и положен в основу входного языка системы Клу-Эльбрус, опубликован в 1981 г.

Основные элементы языка Клу рассмотрим на примере, в котором описывается и используется параметризованный АТД.

```
% описание кластера list
list = cluster [t : type] is create, car, cdr, cons, elems
       rep = array [t]
       create = proc (   ) return (cvt)
                   return (rep $ new (   ))
                   end create;
       car = proc (l : cvt) returns (t) signals (empty)
                 if rep $ empty (l)
                 then signal empty
                 else return (rep $ bottom (l))
                 end
              end car;
       cdr = proc (l : cvt) returns (cvt) signals (empty)
                 if rep $ empty (l)
```

```
              then signal empty
              else
                rep $ reml (l);
                return (l)
              end
              end cdr;
    cons = proc (x : t, l : cvt) returns (cvt)
                rep $ addl (l, x);
                return (l)
              end cons;
    elems = iter (l : cvt) yields (t)
                for elt : t in rep $ elements (l) do
                    yield (elt)
                end
              end elems
  end list;
% использование кластера list
  use = proc ( )
       li = list [int] ; % тип "список целых чисел"
       l : li := li $ create ( );
       l := li $ cons (1, li $ cons (2, li $ cons (3, l)));
       for i: int in li $ elems (l)
     do writeint (i)
     end
  end use
```

Клу – язык модульного программирования. Программа на языке Клу состоит из группы *модулей* – *процедур* (абстракций исполнения), *кластеров* (*абстракций данных*) и *итераторов* (*абстракций управления*). В примере представлены два фрагмента программы – описание кластера *list* и описание процедуры *use*, использующей этот кластер. Кластер *list* определяет класс родственных АТД – список с произвольным типом элементов *t* (*параметром* кластера). Над АТД данного класса определяются *операции*: *create* – создание пустого списка; *car*, *cdr*, *cons* – традиционный набор лисповских операций над списками; *elems* – *итератор* элементов списка. Конкретным представлением списка (**rep**) является массив элементов типа *t*. В языке Клу массив может динамически расширяться и сокращаться с обоих концов, поэтому границы массива в обозначении типа не указываются. Интерфейс и реализация каждой операции кластера *list* описаны совместно.

Операция *create* – процедура без *аргументов*, с *результатом* типа *list*. Однако тип результата обозначен **cvt** (от **convert** – преобразовать), что означает, что внутри операции *create* результат задается объектом типа **rep**, т.е. конкретным представлением, а при выдаче преобразуется к типу *list*. "Внутренний" результат процедуры, определяемый оператором **return**, – пустой массив, созданный операцией *new* из встроенного параметризованного кластера **array** [*t*]. Смысл обозначения **rep** $ *new* – операция *new* из типа **rep** (т.е. **array** [*t*]). Все стандартные типы считаются кластерами, стандартные конструкторы типов (например, **array**) – параметризованными кластерами с определенным набором операций.

Особенности терминологии языка Клу: *t* – *параметр* кластера *list*, *l* – *аргумент* процедуры *cdr*.

Операция *car* выдает значение первого элемента списка *l*. Идентификатор аргумента *l* обозначает *ссылку на объект* (в языке Клу все аргументы передаются способом, аналогичным передаче по ссылке). Тип аргумента **cvt** указывает, что внутри процедуры вместо абстрактного объекта *l* (списка) рассматривается его конкретное представление (массив). Если массив пуст (значение операции **rep** $ *empty* истинно), то процедура *car* завершается возникновением ситуации *empty* (**signal** *empty*), которая должна быть обработана в процедуре-пользователе. Ситуация *empty* указана в заголовке процедуры *car*. Если массив непуст, то в качестве результата выдается значение первого элемента массива, вычисляемое функцией **rep** $ *bottom*.

Операция *cdr* как и операция *car*, сигнализирует о пустом списке ситуацией *empty*. Реализация этой операции иллюстрирует *объектно-ориентированный подход*, принятый в семантике языка Клу. Массив считается *изменяемым объектом*, состояние которого может измениться при выполнении операций. Операция *reml* удаляет из массива и выдает в качестве результата его самый левый элемент (результат операции в данном случае игнорируется). Оператор **return** определяет "внутренний" результат операции *cdr* – укороченный массив *l*. Таким образом, операция *cdr* при данном конкретном представлении влияет на состояние своего аргумента, заменяя его остатком списка. Возможна и другая реализация операции *cdr*, которая выполняла бы *копирование* аргумента.

Операция *cons* реализуется с помощью операции *addl*, которая присоединяет к массиву слева заданный элемент, т.е. выполняет действия, обратные операции *reml*.

Операция *elems*, в отличие от остальных операций, является *итератором* с аргументом типа *list*. В реализации ее используется встроенный итератор *elements*, выдающий элементы массива. Оператор **yield** предназначен для выдачи очередного элемента объекта. Соответственно вместо слова **returns** в заголовке итератора типы выдаваемых им объектов помечаются словом **yields**.

В процедуре *use* используется кластер *list* и некоторые его операции. Модули *list* и *use* являются раздельно компилируемыми; каждый из модулей потенциально доступен из любого другого; внешние ссылки разрешаются с помощью *контекста компиляции*, передаваемого транслятору вместе с компилируемым модулем (пп. 3.10, 3.11). Модули не могут быть вложены друг в друга (важное исключение – вложенность операций в кластер); связь между модулями осуществляется только через аргументы, результаты и ситуации (глобальных переменных нет). Информация периода компиляции о модулях хранится в *библиотеке модулей*, которая используется для контроля связей между ними.

Конкретизация *list* [**int**] кластера *list* определяет новый АТД – список целых чисел. Набор его операций совпадает с набором операций, описанных в классе *list*, но тип-параметр *t* везде заменен на **int**. Для типа *list* [**int**] введено сокращенное обозначение *li* с помощью *тождества* (описания константы). В процедуре *use* описана локальная переменная *l* типа *li*. Переменная, как и аргумент процедуры, обозначает ссылку на объект, а присваивание понимается как перестановка ссылки на другой объект. Начальное зна-

чение переменной *l* – ссылка на новый пустой список. Затем ей присваивается ссылка на список (1, 2, 3), сформированный с помощью операции *cons*. В заключение элементы списка *l* просматриваются в цикле, который управляется итератором *li $ elems*, и печатаются библиотечной процедурой *writeint*.

Средства ввода-вывода языка Клу [83], ввиду ограниченного объема настоящей работы, здесь не рассматриваются, хотя описанием языка [83] предусмотрена мощная система операций *двоичного, текстового* и *терминального ввода-вывода*, имеющая статус стандартного расширения, которая полностью реализована в системе Клу-Эльбрус (автор реализации – А.Е. Соловьев). В примерах используются более простые процедуры *writeint* (*i*) – вывод целого числа *i*, *writereal* (*r*) – вывод вещественного числа *r*; *writestring* (*s*) – вывод строки *s*; *writeln* – перевод строчки при выводе.

Лексика и синтаксис языка Клу имеют следующие особенности. Служебные слова (**cluster**, **proc** и т.д.), как в языке Паскаль, графически неотличимы от идентификаторов и никакими символами не выделяются. Идентификатор может содержать кроме букв и цифр символ подчеркивания, например: *get_element_value*. Точки с запятой, разделяющие описания и операторы, не обязательны и могут ставиться лишь для наглядности. После описания модуля должен повторяться его идентификатор: **end** *list*. Полный синтаксис языка Клу приведен в приложении 3.

3.3. Встроенные типы и генераторы типов

Встроенными в языке Клу называются стандартные (предопределенные) типы и генераторы типов. Каждый из них рассматривается как *кластер* с определенным набором операций. Такой ортодоксальный подход приводит к тому, что каждая стандартная операция должна записываться как вызов операции из кластера: например, сложение двух целых чисел x и y – как **int** *$ add* (x, y). Поэтому для наиболее распространенных операций сохранена их традиционная инфиксная форма: $x + y$. Более того, тот же принцип распространяется и на определяемые типы, вводимые программистом с помощью описаний кластеров: если такой тип содержит двуместную операцию *add* (a, b), то она может быть записана в инфиксной форме: $a + b$. Это сближает определяемые типы со встроенными. Дополнительное сходство придает наличие у стандартных типов общих операций: *create* – генератор объектов; *equal* – совпадение объектов (инфиксная форма: $x = y$); *similar* – структурная идентичность объектов (сходство их поведения; *copy* – копирование объекта. Эти же операции рекомендуется описывать и для определяемых типов.

3.3.1. Встроенные типы. Совокупность встроенных типов языка Клу включает как традиционные типы **int**, **real**, **bool**, **char** и **string**, так и некоторые нововведения **null** ("пустой" тип – "заполнитель места") и **any** (средство ограниченной динамики в Клу – аналог **ф64** в языке Эль-76).

Объекты типов **int**, **real**, **bool**, **char**, **string** и **null** (эти типы будем для краткости именовать *простыми* по аналогии с языком Паскаль) считаются *постоянными*: никакая операция над ними не может изменить состояние (структуру и значение) объекта, а может лишь привести к созданию *нового* объекта с другим состоянием. Над простыми типами не опре-

делена явная операция *create*; новый объект простого типа задается *изображением*: 10 – объект типа **int**, 2.71e–2 – типа **real**; **true** и **false** – типа **bool**; 'a' – типа **char**; "*the clu language*" – типа **string**; **nil** – типа **null**. Изображения объектов типа **char** (символов) заключаются в *апострофы*, строк – в *двойные кавычки*. Над всеми простыми типами определена операция *equal*, *similar* и *copy*. Первые две обозначают совпадение объектов, последняя – тождественная.

Числовые типы **int** и **real** имеют обычный набор операций: *add* (инфиксная форма +), *sub* (–), mul (*), *div* (/; для целых – *деление нацело*), *minus* (унарный минус: –), *power* (возведение в степень: * *), *abs* (модуль), *max*, *min*, *equal* (=) *lt* (<), *le* (<=), *ge* (>=), *gt* (>). Вместо $a <= b$ для чисел можно писать $a \sim > b$. Для нечисловых типов эти обозначения не синонимичны. Для типа **int** определена операция *mod* (// – остаток) и две операции-итератора: **int** $ *from_to_by* (*m*, *n*, *h*) – от *m* до *n* с шагом *h*; **int** $ *from_to* (*m*, *n*)–от *m* до *n* с шагом 1. Для типа **real** определены операции *exponent* и *mantissa* – выделение порядка и мантиссы; *r2i* – округление до целого; *i2r* – обобщение целого до вещественного; *trunc* – целая часть. Из нетрадиционных операций отметим операции *parse* и *unparse*, определенные для обоих числовых типов: *parse* – перевод символьного представления числа (**string**) во внутреннее (**int**, **real**), *unparse* – обратная операция.

Большинство операций над числами может завершиться возникновением *ситуаций*: *overflow* – переполнение, *underflow* – потеря значимости, *zero_divide* – деление на нуль.

Тип **bool** содержит два постоянных объекта – **true** и **false** и логические операции: *and* (инфиксная форма – &), *or* (|), *not* (~). Выражения $a \sim = b$, $a \sim > b$ и т.п. считаются сокращениями выражений: $\sim(a = b)$ и $\sim(a > b)$.

Пример.

Вычисление наибольшего общего делителя *x* и *y*.

```
while y ~= 0 do
   x, y := y, x//y
end;
gcd := x
```

Здесь: $y \sim = 0$ – сокращенное обозначение композиции операций: **bool** $ *not* (**int** $ *equal* (*y*, 0)); тело цикла – *одновременное присваивание* (выражения-правые части исполняются до присваивания).

Символьные типы. Для символьной обработки в языке Клу введены два простых типа – **char** (символ) и **string** (строка). Над типом **char** определены операции отношения *equal*, *lt*, *le*, *ge*, и *gt* и операции преобразования кода символа в символ (*i2c*) и символа в его код (*c2i*). Кодировка символов в реализации Клу, согласно описанию языка [83], должна быть расширением кодировки ASCII. В системе Клу-Эльбрус для кодировки символов используется стандартный код ДКОИ-8.

Тип **string** предназначен для работы со строками. Строка – последовательность символов $s_1 \ldots s_n$, рассматриваемая как постоянный объект. Символы в строке нумеруются с 1.

Операция **string** $ *fetch* (*s*, *i*) выдает значение *i*-го символа (инфиксная форма операции: *s*[*i*]). *Длина* строки вычисляется операцией *size*; операция *empty* – предикат проверки строки на нулевую длину. Для выборки *подстрок* предназначены операции: **string** $ *substr* (*s*, *i*, *k*) – часть строки $s_i s_{i+1} \ldots s_{i+k-1}$ (в терминах языка Эль-76 – *s*[*i* : *k*]) и **string** $ *rest* (*s*, *i*) – часть строки $s_i s_{i+1} \ldots s_n$ (на языке Эль-76 – *s*[*i* :]).

Имеются две операции *поиска по строке*: **string** $ *index* (*c*, *s*) – индекс первого вхождения символа *c* в строку *s* (при отсутствии вхождения результат равен нулю); **string** $ *indexs* (*s1*, *s2*) – индекс первого вхождения строки *s1* в строку *s2* (при отсутствии вхождения результат равен нулю, если строка *s1* пустая – единице).

Операции **string** $ *concat* (*s1*, *s2*) (инфиксная форма – *s1*||*s2*) и **string** $ *append* (*s*, *c*) характерны для символьной обработки: первая выдает конкатенацию строк *s1* и *s2*, вторая – конкатенацию строки *s* и символа *c*.

Над строками определены также традиционные операции *лексикографического порядка* *lt* (<), *le* (<=), *ge* (>=), *gt* (>) и три операции *преобразования*: *c2s* – символа в односимвольную строку; *s2ac* – строки в массив символов (изменяемый объект типа **array**[**char**]), *s2sc* – строки с последовательность символов (постоянный объект типа **sequence** [**char**]). Имеются и обратные операции: *ac2s* обратна *s2ac*, *sc2s* обратна *s2sc*.

Итератор *chars* выдает символы строки в порядке их индексов.

При выборке элемента или подстроки могут возникнуть ситуации *bounds* (выход индекса за границу строки) и *negative_size* (размер подстроки отрицателен).

Пример.

Процедура контекстного редактирования – замена всех вхождений строки *s1* в строку *s* на строку *s2*. Результат – строка, полученная после редактирования. Строка *s1* предполагается непустой.

```
edit = proc (s, s1, s2: string) returns (string)
        size = string $ size; index = string $ indexs;
        cur : string := s;
        res : string := "" ;
        s1_size : int := size (s1);
        cursize : int := size (cur);
        i : int := index (s1, cur);
        while i ~= 0 & cursize ~= 0 do
            if i ~= 1 then res := res || string $ substr (cur, 1, i)
            end;
            res := res || s2;
            cur := string $ rest (cur, i + s1_size) ;
            cursize := size (cur);
            i := index (s1, cur)
        end;
        if cursize ~= 0 then res := res || cur end;
        return (res)
end edit
```

Тип **null** имеет единственное значение **nil** и операции *equal*, *similar* и *copy*. Он используется как "заполнитель места" в объектах-вариантах (п. 3.3.7) и объединениях (п. 3.3.6).

Тип any введен в язык Клу как средство ограниченной динамики, необходимой для разработки системных программ. Обозначение **any** указывает, что тип переменной, элемента массива и т.п. может быть произвольным. Например, спецификация **any** в описании переменной *v*

v : any

означает, что переменной *v* могут быть присвоены значения любых типов:

v := 1; . . .
v := "*string*"

В этом отношении переменная типа **any** аналогична переменной формата **ф64** в языке Эль-76. Информация о текущем типе переменной *v* хранится динамически и является аналогом тега. Однако в отличие от Эль-76, Клу – язык со статической типизацией. Поэтому при использовании значения *v* в операциях должен быть указан его предполагаемый тип:

force [**string**] (*v*) || "*строка 1*"

Операция **force** [*t*] – встроенный генератор процедур – выполняет динамическую проверку типа значения переменной типа **any**. Если он совпадает с *t*, то выдается фактическое значение типа *t*, иначе возникает ситуация *wrong_type* (неверный тип).

Над самим типом **any** не определено никаких операций.

3.3.2. Массивы. Массив – это объект типа **array** [*t*], где **array** – встроенный генератор типов (параметризованный кластер), *t* – тип элементов. Массивы в языке Клу отличаются динамичностью своей структуры. Помимо обычной возможности изменения элементов массив можно расширять и сокращать с обоих концов и менять в нем нумерацию элементов. Все эти модификации трактуются как изменение состояния исходного массива, а не как создание нового массива или ссылки на подмассив. *Состояние* массива определяется текущей *нижней границей* и последовательностью *элементов*, нумеруемых начиная с этой нижней границы.

Наиболее лаконичный по форме способ создания массива – *конструктор массива*:

ai = **array** [**int**] ;
a : *ai* := *ai* $ [1 : 15, 20, 25]

Массив *a* имеет нижнюю границу 1 и целые элементы: *a*[1] = 15, *a*[2] = = 20, *a*[3] = 25. Таким образом, созданный элемент получает некоторое значение, и неопределенных элементов в массиве нет. Операция *ai* $ *create*(5) создает пустой массив с границей 5, *ai* $ *new* ( ) – пустой массив с нижней границей 1 (в операции "конструктор массива" операция *create* выполняется неявно). Операция *ai* $ *fill* (1, 100, *x*) создает новый массив из 100 элементов, равных *x* (нижняя граница – 1). Операция *fill_copy* (аргументы – те же) формирует новый массив из элементов с одинаковыми состояниями.

Для выборки и изменения элемента массива *a* служат операции *ai* $ *fetch* (*a*, *i*) и *ai* $ *store* (*a*, *i*, *x*). В более привычной форме, введенной как сокращение, они записываются как *a*[*i*] и *a*[*i*] := *x* Операции *bottom* и *top* выдают первый и последний элементы массива.

Операции *low*, *high* и *size* вычисляют соответственно нижнюю и верхнюю границы индексов и длину массива. Операция *empty* осуществляет проверку на пустой массив. Нижняя граница индекса может быть изменена операцией *ai* $ *set_low*(*a*, *m*). При этом происходит перенумерация всех элементов от *m*.

Операции *addl* и *addh* расширяют массив снизу и сверху, операции *reml* и *remh* – сокращают. При этом сохраняемые элементы сохраняют и свои индексы. Например, операция *ai* $ *addl*(*a*, 10) добавляет к массиву элемент *a* [0] = 10; операция *ai* $ *addh*(*a*, 30) добавляет элемент *a* [4] = 30. Если массив *a* в первоначальном состоянии, то операция *ai* $ *reml*(*a*) уничтожает элемент с индексом 1 и выдает его значение 15 в качестве результата. Аналогично *ai* $ *remh*(*a*) = 25.

С операциями *addh* и *addl* связана еще одна форма генератора массива. Операция *ai* $ *predict* (1, 1000) создает пустой массив с нижней границей 1 и обеспечивает более эффективное выполнение операций изменения длины массива *addh* (*addl*) от 0 до 1000, т.е. резервирует место для 1000 элементов.

Аналогом вырезки является операция *trim*. Однако она не создает ссылку на часть массива, а сокращает сам массив: операция *ai* $ *trim*(*a*, 2, 1) устанавливает массив равным его отрезку начиная со второго элемента, длиной 1, т.е. после ее выполнения массив *a* будет состоять из одного элемента *a* [2] = 20.

Над массивами определено две операции-итератора: *elements* выдает элементы массива в порядке возрастания индексов, *indexes* – сами значения индексов в порядке их возрастания.

Набор операций сравнения и копирования следующий. Операция *equal* проверяет два объекта-массива на совпадение (результат истинен, если операнды ссылаются на один объект), *similar* – на идентичность состояний (совпадение индексов, длин и подобие элементов в смысле операции *similar*); *similar1* – то же, что и *similar*, но над элементами проверяется отношение *equal*. Операция *copy* создает новый массив с теми же элементами, что и исходный; *copy* – с элементами-копиями элементов исходного массива (фактически массив в языке Клу – это массив ссылок на элементы).

В операциях над массивами могут возникнуть ситуации *bounds* (выход за границу массива) и *negative_size* (аргумент операции – отрицательный размер).

Пример.

Конкатенация двух массивов целых чисел. Нижняя граница результата равна нижней границе первого массива.

```
ai = array [int];
ai_concat = proc (a1, a2 : ai) returns (ai)
   size1 : int := ai $ size (a1);
   size2 : int := ai $ size (a2);
   res : ai := ai $ predict (ai $ low (a1), size1 + size2);
   for elt1 : int in ai $ elements (a1) do
      ai $ addh (res, elt1)
   end;
```

```
    for elt 2 : int in ai $ elements (a2) do
        ai $ addh (res, elt2)
    end;
    return (res)
end ai_concat
```

В примере для перебора элементов массивов *a*1 и *a*2 вместо итератора *elements* можно было бы использовать и итератор *indexes*:

```
for i : int in ai $ indexes (a1) do
    ai $ addh (res, a1 [i ]) end
```

3.3.3. Последовательности. Последовательность – это постоянный массив. Типы последовательностей задаются встроенным генератором типов **sequence** [t]. Объект типа "последовательность", в отличие от массива, постоянен. Элементы последовательности нумеруются всегда начиная с 1.

Многие операции над последовательностями совпадают по имени и по семантике с операциями над массивами (п. 3.3.2); *new* – создание пустой последовательности; *size* – длина последовательности; *empty* – проверка на пустую последовательность; *fill* и *fill_copy* – создание последовательности из одинаковых и подобных элементов; *fetch* – выборка элементов (инфиксная форма – *s* [*i*]); *elements* – итератор элементов; *indexes* – итератор индексов; *equal* – поэлементное равенство; similar – поэлементное сходство; *copy* – копирование. Операции *addh*, *addl*, *remh* и *reml* по форме аналогичны операциям над массивами, но отличаются по семантике: каждая из них создает новую последовательность с соответствующим состоянием, перенумеровывает ее элементы с единицы и выдает ее в качестве результата. Вместо операции изменения элемента *store* для последовательностей определена операция *replace*(*s*, *i*, *x*), выдающая новую последовательность со значением *x* элемента *s* [*i*].

Специфические операции над последовательностями:

subseq (*s*, *i*, *k*) – подпоследовательность элементов с индекса *i* длиной *k* (на языке Эль-76 – *s* [*i* : *k*]);

concat (*s*1, *s*2) – конкатенация последовательностей *s*1 и *s*2 (инфиксная форма – *s*1 || *s*2);

*a*2*s*(*a*) – преобразование массива *a* в последовательность;

*s*2*a*(*s*) – преобразование последовательности *s* в массив;

*e*2*s*(*x*) – преобразование *x* в одноэлементную последовательность.

В операциях над последовательностями могут возникать те же ситуации *bounds* (выход за границу и *negative_size* (отрицательный размер), что и в операциях над массивами).

Пример.

Слияние упорядоченных последовательностей.

```
sr = sequence [real];
merge = proc (s1, s2 : sr) returns (sr);
    res : sr := sr $ new ( );
    i1 : int := 1;
    i2 : int := 1;
    size1 : int := sr $ size (s1);
    size2 : int := sr $ size (s2);
```

```
while i1 <= size1 & i2 <= size2 do
   if s1 [i1]<s2 [i2]
   then
      res := sr $ addh (res, s1 [i1]);
      i1 := i1 + 1
   else res := sr $ addh (res, s2[i2]);
      i2 := i2 + 1
   end
end;
if i1<=size1
then res := res || sr $ subseq (s1, i1, size1–i1 + 1)
elseif i2 <= size2
then res := res || sr $ subseq (s2, i2, size2 – i2 + 1)
end;
return (res)
end merge
```

3.3.4. Записи. Понятие записи в языке Клу по своей форме и набору операций достаточно традиционно, однако авторы языка трактуют запись, аналогично массиву, как встроенный генератор типа (параметризованный кластер). Генератор типов-записей имеет вид

record $[f_1 : t_1, \ldots, f_n : t_n]$

где f_i – *имена полей*, t_i – обозначения их *типов*. Таким образом, тип-запись зависит не только от типов полей, но и от их *имен*. Кроме того, число параметров генератора типов **record**, т.е. число полей, не является постоянной величиной. Поэтому, как отмечается в работе [34], генератор типа **record**, строго говоря, не является кластером. Однако эта формальная сложность не приводит к каким-либо неудобствам в практике программирования, кроме отсутствия итераторов *fields* (поля) и *names* (имена полей) для покомпонентного анализа записей.

Запись, как и массив, является *изменяемым объектом*. Создание нового объекта-записи выполняется *конструктором записи*, в котором указываются имена и значения всех полей:

```
complex = record [ re, im : real ];
z : complex := complex $ {re : 1.0, im : 2.0}
```

Здесь конструктор записи образует правую часть оператора присваивания.

Основные операции над записями – *выборка* и *изменение* значения *поля*. В языке Клу они определены в форме: *rt $ get_f(r)* и *rt $ set_f(r, x)*, где *rt* – тип-запись, *f* – имя поля, *r* – объект-запись, *x* – новое значение поля. Эти операции имеют и традиционные инфиксные формы: *r.f* и *r.f := x*.

Аналог *поэлементного присваивания* записей реализован в виде операции *rt $ r_gets_r(r1, r2)*, где *r1* и *r2* – записи типа *rt*. Операция заменяет значения компонент записи *r1* значениями компонент записи *r2*. Имеется также аналогичная операция *rt $ r_gets_s(r, s)*, заменяющая значения компонент записи компонентами *структуры s*, имеющей такие же поля (п. 3.3.5).

Для записей определены операции *сравнения* и *копирования*: *equal* – *совпадение* объектов-записей; *similar* – *сходство* записей (совпадение имен полей и выполнение отношения *similar* над полями); *similar*1 – то же, что *similar*, но для полей проверяется отношение *equal*;*copy*1 – *копирование* записи *без копирования полей*; *copy* – *копирование* записи *с копированием полей*.

Пример.

Действия над комплексными переменными ($z1$ присваивается сумма чисел $z1$ и $z2$).

complex = **record** [*re*, *im* : **real**];
*z*1 : *complex* := *complex* $ {*re* : 1.0, *im* : 2.0};
*z*2 : *complex* := *complex* $ { *re* : 5.0, *im* : 3.0};
complex $ *r_gets_r* (*z*1, *complex* $ { *re* : *z*1.*re* + *z*2.*re*, *im* : *z*1.*im* + *z*2.*im*})

В примере использование операции *r_gets_r* обеспечивает поэлементное присваивание объекту $z1$ без создания нового объекта.

3.3.5. Структуры. Структура – это *постоянная запись*. Типы объектов-структур аналогично записям задаются *генератором типа*

struct [f_1 : t_1, . . . ,f_n : t_n]

Основные операции над структурой – *конструктор структуры* и *выборка поля* – аналогичны операциям над записями. Вместо операции *set_f* введена операция *replace_f*(s, x), результатом которой является новая структура со значением поля f, замененным на x.

Операция $s2r$ преобразует структуру в запись с теми же компонентами. Обратная операция – $r2s$.

Операция сравнения equal выдает значение **true**, если все соответствующие компоненты совпадают (т.е. над ними выполняется отношение *equal*); *similar*, если над компонентами выполнено отношение *similar*. Операция *copy* создает *копию* структуры из копий компонент.

Пример.

Обработка анкет.

person = **struct** [*name*, *surname* : **string**, *age* : **int**];
p : *person* := *person* $ { *surname* : "*shaw*", *name* : "*mary*", *age* : 30};
p := *person* $ *replace_surname* (*p*, "*pickford*")

3.3.6. Объединения. *Типы-объединения* языка Клу аналогичны объединенным типам (**union**) языка Алгол-68 и, как и тип **any**, являются средством ограниченной динамики. Форма *генераторов типов-объединений* сходна с генератором типов-записей:

oneof [f_1 : t_1, . . . ,f_n : t_n] ,

но компоненты имеют другой смысл. *Объединение* – это постоянный объект, состоящий из *признака* (символьного имени) f и *объекта* некоторого типа t. Возможные значения признака задаются именами f_i, соответствующие им типы объектов – обозначениями типов t_i. Например, структура машинного слова с тегом, содержащего либо неопределенную информацию, либо объект одного из простых типов **int**, **real**, **bool**, **char**, **string**, описывает-

ся следующим типом-объединением:

$word$ = **oneof** [$empty_val$: **null**, int_val : **int**, $real_val$: **real**,
$bool_val$: **bool**, $char_val$: **char**, $string_val$: **string**]

Основные операции над объединениями связывают две их компоненты – признак и значение. Операция $make_f(x)$ создает объект типа "объединение" u с признаком f и значением x типа t. Операция $is_f(u)$ проверяет, обладает ли объединение u признаком f. Операция $value_f(u)$ осуществляет выборку объекта-компоненты из объединения u, если его признак равен f, иначе возникает ситуация *wrong_ tag* – неверный признак.

Операция *equal* сравнивает объединения на совпадение признаков и объектов-компонент, операция *similar* – на совпадение признаков и подобие компонент (в смысле операции *similar*). Операция *copy* создает копию объединения из копии его компоненты.

Операция $o2v$ и $v2o$ осуществляет связь с другой (изменяемой) формой объединений – *вариантами* (п. 3.3.7): $o2v$ преобразует объединение в вариант, $v2o$ – вариант в объединение.

Пример.

Работа со словами.

$w1$: $word$:= $word$ ¢ $make_int_val(1)$;
$w2$: $word$:= $word$ ¢ $make_int_val$ (2);
$w3$: $word$:= $word$ ¢ $make_real_val$ (3.0);
i : **int** := $word$ ¢ $value_int_val$ ($w1$) +
$word$ ¢ $value_int_val$ ($w2$) –
real ¢ $r2i$ ($word$ ¢ $value_real_val$ ($w3$))

3.3.7. Варианты. Вариант – это изменяемый объект, аналогичный объединению по структуре и набору операций. Обозначение типа "вариант" имеет вид

variant $[f_1 : t_1, \ldots, f_n : t_n]$,

где, как и для объединения, f_i – *имена признаков*, t_i – типы соответствующих им *объектов*. Состояние объекта-варианта – это пара (f, x), где f – имя признака, x – объект типа t.

Основные операции над вариантами – те же, что и над объединениями: $make_f(x)$ – создание варианта с признаком f и объектом x (последний должен иметь соответствующий тип); $is_f(v)$ – проверка состояния варианта v на признак f; $value_f(v)$ – выборка объекта-компоненты варианта v при условии, что значение признака равно f (иначе – ситуация *wrong_tag*. Поскольку вариант – изменяемый объект, введена операция изменения его состояния $change_f(v, x)$ (присваивает варианту признак f и объект x).

Над вариантами определены две операции *покомпонентного присваивания*: $v_gets_v(v1, v2)$ – присваивание варианту $v1$ компонент варианта $v2$; $v_gets_o(v, o)$ – присваивание варианту v компонент объединения o.

Как и над другими изменяемыми объектами, над вариантами определено пять операций *сравнения* и *копирования*: *equal* – *совпадение* объектов; *similar* – совпадение признаков и подобие объектов-компонент (в смысле отношения *similar*); *similar*1 – то же, что *similar*, но над компонентами про-

веряется отношение *equal*; *copy* – создание нового варианта с копией объекта-компоненты заданного варианта; *copy*1 – создание копии объекта-варианта с тем же объектом-компонентой, что и у исходного варианта.

Пример.

Моделирование векторов формата ф64 МВК "Эльбрус" и операции их пересылки с тегами (1.6.4).

```
word =  variant [empty : null, int_val : int, real_val : real,
        bool_val : bool, char_val : char, string_val : string];
 vector = array [word];
 assign_vector = proc (v1, v2) returns (bool, bool)
s1 : int := vector $ size (v1);
 s2 : int := vector $ size (v2);
 m : int := int $ min (s1, s2);
 l1 : int := vector $ low (v1);
 l2 : int := vector $ low (v2);
 for i : int : in int $ from_to (0, m - 1) do
     word $ v_gets_v (v1 [i + l1], v2[i + l2])
 end;
 return (s1 <= s2, s1 >= s2)
end assing_vector
```

Процедура *assing_vector* выдает два логических результата, которые соответствуют значениям признаков тгп и тги (п. 1.6.4).

3.3.8. Типы процедур и итераторов. Процедуры и итераторы – модульные объекты, создаваемые клу-системой. Как и другие объекты, они имеют определенные типы. Обозначение *типа процедуры* имеет следующий вид:

proctype $(a_1, \ldots, a_n)$ **returns** $(r_1, \ldots, r_m)$
signals $(s_1, \ldots, s_k)$

где a_i – типы аргументов; r_j – типы результатов; s_l – имена ситуаций, которые могут возникнуть при исполнении процедуры, и типы их возможных аргументов (п. 3.6). Аналогичную форму имеет *тип итератора* (п. 3.7):

intertype $(a_1, \ldots, a_n)$ **yields** $(r_1, \ldots, r_m)$ **signals** $(s_1, \ldots, s_k)$

Здесь r_j – типы объектов, выдаваемых при выполнении итераций. Если аргументы отсутствуют, они заменяются пустым списком ( ). При отсутствии результатов или ситуаций компонента **returns** (**yields**) или **signals** опускается.

Например, операция *int* $ *lt* имеет тип

proctype (int, int) returns (bool)

а операция **array** [t] $ *store* – тип

proctype (array [*t*], **int**, t) **signals** (*bounds*)

Тип операции-итератора **int** $ *from_to_by* задается обозначением

itertype (int, int, int) yields (int)

Над процедурами и итераторами определены операции сравнения *equal* и *similar*. Обе они выдают результат **true** в том и только в том случае, если операнды – процедуры или итераторы обозначают одну и ту же реализацию одного и того же модуля, а если он параметризован, то одну и ту же его конкретизацию. Операция *copy* для процедур и итераторов определена как тождественная.

Вызов процедуры (итератора), согласно семантике языка Клу, является не операцией, а одним из базовых действий клу-системы.

3.4. Типы, объекты, константы и переменные

В этом параграфе описаны основные особенности семантики языка Клу, связанные с понятиями типа, объекта, константы и переменной. Рассмотрим также различные формы описаний.

3.4.1. Объекты и переменные. Объект – это элемент данных, создаваемый и обрабатываемый программой на языке Клу. Объект характеризуется своим *состоянием* – структурой и значениями компонент и *поведением* – изменением состояния при выполнении операций. Набор операций, применимых к объекту, задается в определении типа этого объекта.

Объекты подразделяются на *постоянные* – с неизменным состоянием – и *изменяемые* – состояние которых может модифицироваться при выполнении операций. Примеры постоянных объектов – число, строка, последовательность. В операциях над постоянными объектами вместо изменения их состояния выполняется создание новых объектов с модифицированным состоянием. Например, операция *addh* над массивом изменяет состояние самого массива, добавляя к нему сверху новый элемент; эта же операция над последовательностями создает *новую* последовательность, в которой на один элемент больше.

Ссылка на объект осуществляется либо через другой объект, либо с помощью *переменной* или *аргумента* модуля. Например, массив ссылается на свои элементы (т.е. фактически состоит из ссылок на объекты-элементы, а не из самих элементов). Новая переменная, локальная в некотором блоке, создается путем описания:

$x : t$

внутри этого блока. Переменная x может ссылаться на объекты типа t, например:

$x := t \$ create\ (\ )$

Таким образом, любая переменная в языке Клу неявно считается *ссылочной*, а *присваивание* понимается как *установка ссылки* (в форме переменной) на заданный объект. Присваивание

$x1 := x$

устанавливает ссылку из переменной $x1$ на *тот же объект*, на который ссылается и переменная x. Аргумент процедуры, как и переменная, обозначает ссылку на объект. Например, при исполнении процедуры

```
p = proc (v1, v2 : array [int])
    v1 [1] := 0;  v2 [2] := 0
end p
```

в результате ее вызова

$p(a, a)$

аргументы $v1$ и $v1$ *разделяют* (ссылаются на) один и тот же *объект* (массив a), и в результате исполнения процедуры p:

$a[1] = a[2] = 0.$

Любой созданный объект является *глобальным* и существует до тех пор, пока на него имеются ссылки.

3.4.2. Обозначения и контроль типов. Клу – язык с полным статическим контролем типов. Любой объект, переменная или аргумент имеют вполне определенный *тип*, который определяет набор допустимых *операций* над объектом. Каждый тип в языке Клу рассматривается как *кластер* (АТД) – либо *встроенный* (п. 3.3), либо *определяемый* программистом. Таким образом, структура (конкретное представление) объектов любого типа скрыта от программиста, использующего данный тип, а ему доступны лишь операции, входящие в *интерфейс* типа.

В программе тип задается *обозначением типа*, которое представляет собой либо служебное слово, обозначающее встроенный тип (**int**, **real** и т.д.), либо идентификатор определяемого типа, либо конкретизацию параметризованного кластера (встроенного или определяемого). Если c – кластер:

c = **cluster is** $op_1, \ldots, op_n$ ...

а cc – параметризованный кластер:

cc = **cluster** [t : **type**] **is** ... ,

то примерами обозначений типов являются:

array [c] – массив объектов типа c;

sequence [cc [**int**]] – последовательность объектов типа cc [**int**];

cc [**record** [x : c; y : **sequence** [**string**]]] – конкретизация параметризованного кластера cc типом записи с полями x типа c и y типа "последовательность строк".

Над обозначениями типов определено отношение эквивалентности, разделяющее множество обозначений типов на классы такие, что два представителя одного класса обозначают один и тот же тип. Эквивалентными считаются:

– служебные слова или идентификаторы, обозначающие один и тот же кластер;

– идентификаторы типов, обозначающие эквивалентные типы;

– конкретизация одного и того же кластера с эквивалентными параметрами-типами и равными параметрами-константами;

– две формы обозначения типа **record**, **struct**, **oneof** или **variant** с эквивалентными типами компонент, отличающиеся только порядком и группировкой полей; например, эквивалентными считаются обозначения типов

record [a, b : **int**, c : **array** [**real**]]

и

record [c : **array** [**real**], a : **int**, b : **int**]

— две формы обозначения типа **proctype** или **itertype**, с эквивалентными типами компонент, отличающиеся только порядком имен ситуаций; например, обозначения типов

proctype (**int**) **returns** (**char**) **signals** (*overflow*, *underflow*)

и

proctype (**int**) **returns** (**char**) **signals** (*underflow*, *overflow*)

Отметим, что такая эквивалентность обозначений типов отличается от понятия структурной идентичности типов (п. 2.3.1) и, в частности, не приводит к нарушению инкапсуляции. В языке Клу (в отличие от языка Паскаль и Алгол-68) обозначение типа задает не его конкретное представление, а абстрактную логическую организацию. Конкретное представление объекта всегда скрыто внутри кластера, а доступ к объекту осуществляется только через заданные операции, совокупность которых не зависит, например, от порядка полей в обозначении типа "запись". В частности, новый кластер, даже "структурно идентичный" уже существующему, считается новым АТД, не идентичным никакому другому.

При присваивании тип объекта в правой части должен совпадать с типом переменной. Аналогично при передаче аргумента или выдаче результата его тип должен совпадать с типом, указанным в заголовке модуля. Исключение составляет случай, когда переменная, аргумент или результат имеют тип **any**; в этом случае фактический тип объекта может быть любым.

3.4.3. Описания констант и переменных. Для обозначения типов и констант в языке Клу введены *описания-тождества*. Под константой в языке Клу понимается постоянный объект, известный во время компиляции, т.е. изображение объекта простого типа **null**, **int**, **real**, **bool**, **char**, **string**, либо результат применения к ним композиции операций, выдающих объекты *константных типов* (простых типов, последовательностей и объединений). Константами считаются также *имена процедур и итераторов* (в том числе операций кластера).

Пример последовательности тождеств:

sc = **sequence** [**char**];
sc_sub = *sc* $ *subseq*;
*s*1 = *sc* $ ['*a*', '*b*', '*c*', '*d*', '*e*'];
*s*2 = *sc_sub* (*s*1, 2, 3);
b = *s*2 [1]

Первое тождество вводит идентификатор типа *sc*, остальные – идентификаторы констант. Все операции, используемые в правых частях, должны быть выполнимы во время компиляции. В частности, константа *b* имеет тип **char** и значение '*b*'.

Идентификаторы типов и констант определены до конца ближайшего *блока* (замкнутой конструкции с локальными описаниями). В одной последовательности тождеств разрешены ссылки на идентификатор, определяемый позже в этой же последовательности. При этом не допускаются циклические ссылки (рекурсивность типов данных выражается через кон-

кретное представление в описании кластера). Последовательность тождеств

$a = b + c$;
$c = d * e$;
$e = b - 1$;
$b = 2$;
$d = 3$

корректна, а последовательность

elt = **record** [*inf* : **int**, *nxt* : *list*];
list = *elt*

некорректна (сравните с традиционным описанием типа "список" в языке Паскаль). Описания-тождества должны задаваться *в начале блока*.

В некоторых случаях вместо описаний-тождеств следует использовать *описания переменных*, так как обозначаемый объект не является константой (в смысле данного выше определения), хотя при исполнении блока он и не изменяется:

ai = **array** [**int**];
a : *ai* = *ai* $ [1 : 0, 25, 3];
s : **int** = *ai* $ *size*(*a*)

В общем случае описание переменных может иметь вид

$v_1, \ldots, v_n : t$

(описание без инициализации) или

$v_1 : t_1, \ldots, v_n : t_n := p(x_1, \ldots, x_k)$

(описание с инициализацией), где t – обозначение типа; p – процедура, выдающая n результатов, которые становятся значениями переменных $v_1, \ldots, v_n$. Описание переменных может быть задано в любом месте блока и действует до конца блока.

В отличие от обычных алголоподобных языков, в языке Клу переопределение идентификатора во вложенном блоке запрещено.

3.5. Управляющие конструкции

3.5.1. Присваивание – это элементарное действие клу-системы. Присваивание в языке Клу понимается как установка ссылки на объект x (п. 3.4.1) с помощью переменной или аргумента v:

$v := x$

Переменная v должна иметь либо тип, совпадающий с типом x, либо тип **any**. Другая форма присваивания – *групповое* (одновременное) *присваивание*:

$v_1, \ldots, v_n := x_1, \ldots, x_n$

или

$v_1, \ldots, v_n := p(y_1, \ldots, y_k)$

где v_i – переменные или аргументы, x_i, y_j – ссылки на объекты, p – процедура с n результатами. Переменной v_i присваивается либо ссылка x_i, либо i-й результат процедуры p. Правая часть вычисляется до исполнения собственно присваивания. Поэтому групповое присваивание удобно использовать для обмена значениями или других подобных действий, которые обычно требуют дополнительных переменных:

$a, b, c := b, c, a;$

$x, y := x \mid y,\ x \parallel y$

Кроме присваивания, знаком ":=" обозначается также инфиксная форма операций *store* (над массивами) и *set_f* (над записями) (пп. 3.3.2, 3.3.4). Например:

$a[i] := x;$

$r.f := x$

Смысл этих действий аналогичен присваиванию: устанавливается ссылка из компоненты объекта $a(r)$ на объект x.

3.5.2. Оператор "блок" имеет традиционный вид:

begin *тело* **end**

где *тело* – последовательность тождеств, описаний и операторов. Тело может входить в состав любой замкнутой управляющей конструкции и образует самостоятельный блок с локальными описаниями.

3.5.3. Условный оператор записывается в обычной форме; несколько отличается от обычного лишь написание служебных слов:

if b_1 **then** *тело*$_1$
elseif b_2 **then** *тело*$_2$
. . .
elseif b_n **then** *тело*$_n$
else *тело*
end

где b_i – выражение типа **bool**. Компоненты **elseif** и **else** могут отсутствовать.

3.5.4. Оператор цикла существует в двух вариантах: цикл **while** и цикл **for**. Форма цикла **while** традиционна:

while b **do** *тело* **end**

Оператор **for** в языке Клу управляется с помощью *итератора*. Иными словами, в этом операторе объединены как обычные циклы с параметром, принимающим целые значения, так и более сложные формы циклов: просмотр списка, обход дерева и т.п. Оператор **for** имеет вид

for $i_1 : t_1, \ldots, i_n : t_n$ **in** *iterator* $(x_1, \ldots, x_k)$ **do**
тело
end

где $i_1, \ldots, i_n$ – описание переменных цикла типов $t_1, \ldots, t_n$; *iterator* $(x_1, \ldots, x_k)$ – вызов итератора, выдающего n результатов. Тело цикла выполняется, пока итератор продолжает выдавать новые кортежи объектов $i_1, \ldots, i_n$; по завершении итератора цикл заканчивается.

П р и м е р ы.

1. Сумма нечетных элементов массива вещественных чисел

```
for i : int in int $ from_to_by (1, array [real] $ size(a), 2)
do s := s + a [i]
end
```

2. Вычисление максимального значения элемента последовательности:

```
for r: real in sequence [real] $ elements (s)
do if max < r then max: = r end
end
```

Операторы **break** и **continue** служат для управления работой циклов: **break** прекращает исполнение текущего цикла, **continue** завершает исполнение текущей итерации.

П р и м е р.

Суммирование целых элементов числового массива. Массив может содержать целые, вещественные и неопределенные элементы. При обнаружении неопределенного элемента суммирование прекращается.

```
cell = oneof [i: int, r : real, u : null];
ac = array [cell];
a : ac; s : int : = 0;
... % инициализация массива a
for c : cell in ac $ elements (a)
do
   if cell $ is_ r (c) % вещественные
   then continue
   elseif cell $ is _ u (c) % не определено
   then break
   else % целое
        s:= s + cell $ value _ i (c)
   end
end
```

3.5.5. Оператор выбора используется для анализа объединений и вариантов. По форме и назначению он близок к оператору выбора (**case**) по значению типа **union** в Алголе-68. Например, с использованием оператора выбора цикл из примера к п. 3.5.4 можно переписать так:

```
for c : cell in ac $ elements (a)
 do
   tagcase c
     tag i (n : int) : s : = s + n;
     tag u : break
     others : continue
   end
end
```

Оператор выбора является телом цикла. В качестве селектора выступает объект *c* типа *cell*. В операторе **tagcase** перечислены все возможные значения признака, для каждого из которых задано тело реакции. Если значение признака равно *i* (целое), выбирается целая компонента объекта *c* (она

обозначена через *n*). В остальных случаях значение компоненты несущественно. Ветвь **others** обрабатывает все значения признака, отличные от *i* и *u*, так что в этой форме цикл будет правильно работать и в случае, если ячейка *cell* содержит не только числа, но и значения других типов, например символы.

3.5.6. Обработка ситуаций. Смысл и назначение понятия "ситуация" на языке Клу такие же, как в языке Эль-76 (п. 1.15). Однако способ обработки и семантика распространения ситуации несколько иные. Ситуации в Клу (как и в Эль-76) можно условно разделить на статические и динамические. Первые из них используются для завершения управляющих конструкций в пределах одной *подпрограммы* (процедуры или итератора), вторые – для выражения исключительного исхода работы самой подпрограммы. В первом случае ситуация возникает при выполнении оператора **exit** *s* $(x_1, \ldots, x_n)$, во втором – **signal** *s* $(x_1, \ldots, x_n)$, где x_i – возможные аргументы ситуации *s*. В обоих случаях обработка ситуации должна быть выполнена оператором **except** (его форма разъясняется на примерах). Если статическая ситуация *s* не обработана в текущей подпрограмме, то последняя завершается возникновением стандартной ситуации *failure* с выдачей сообщения: "не обработана ситуация *s*". Если динамическая ситуация не обработана в вызывающей подпрограмме, то последняя также завершается ситуацией *failure*.

Пример 1.

Статические ситуации (суммирование элементов массива до нахождения элемента, равного *x*).

```
ai = array [int];
a : ai;
x : int;
s : int := 0;

for i : int in ai $indexes (a) do
   if a [i] = x then exit found (i)
   else s := s + a [i]
   end
end
except when found (k : int) : writeint (k)
       when overflow : s := 0;
                       writestring ("overflow")
       others (s : string) : writestring (s)
end
```

Цикл **for** является частью оператора **except**, в котором заданы реакции на ситуации *found* и *overflow*, а также реакция на все прочие (**others**) ситуации, которые могут возникнуть при исполнении цикла. В реакции **when** *found* описан целый аргумент *k*, так как ситуация *found* в теле цикла возникает с целым аргументом. Если аргументы ситуации при ее обработке несущественны, используется форма

when *found* (*): *тело*

Ситуация *overflow* — *стандартная;* она может возникнуть в операции сложения. Заметим, что ситуация *overflow* трактуется как динамическая, так как операция **int** $ *add* считается отдельной подпрограммой, но на способ ее обработки это не влияет. Обработка прочих ситуаций заключается в том, что через аргумент *s* выдается имя ситуации в виде строки, которое в реакции печатается. Аргумент-строка в реакции **others** обязателен. Если имени ситуации знать не требуется, реакция на прочие ситуации имеет вид: **others**: *тело*. Наконец, она может быть вообще опущена, но это может привести к возникновению ситуации *failure* в текущей процедуре из-за необработанной ситуации.

Пример 2.

Динамические ситуации (определение и вызов процедуры).

```
ar = array [real];
scal = proc (a, b : ar) returns (real)
            signals (overflow, underflow, bounds)
   if ar $ low (a) ~= ar $ low (b) cor
     ar $ high (a) ~= ar $ high (b)
   then signal bounds
   else s : real : = 0.0;
      for i : int in ar $ indexes (a) do
         s : = s + a [i] * b [i]
      end resignal overflow, underflow;
      return (s)
   end end scal;
   . . .
begin % использование scal
x : ar : = ar $ [ 1 : 3.14, – 2.71, 0.0] ;
y : ar : = ar $ [ 1 : 1.57, 0,25, 3.1] ;
s : real : = scal (x, y);
writereal (s)
  end
  except when bounds : writestring ("bounds not equal")
         when overflow: writestring ("overflow")
         when underflow : writestring ("underflow")
  end
```

В заголовке процедуры *scal* указаны все возможные ситуации: *bounds* (границы) — создается оператором **signal** при несовпадении границ векторов-аргументов; *overflow* и *underflow* — стандартные ситуации, которые могут возникнуть в арифметических операциях. Оператор **signal** *bounds* завершает исполнение текущей процедуры (*scal*) и создает ситуацию *bounds* в вызывающем модуле. Область действия ситуаций *overflow* и *underflow*, возникающих во встроенных операциях, ограничена телом процедуры *scal*, но оператор **resignal** (продолжение ситуации) обеспечивает распространение ситуации в модуле, вызывающем процедуру *scal*. Оператор

resignal *overflow, underflow*

эквивалентен последовательности реакций:

```
except when overflow: signal overflow
       when underflow: signal underflow
end
```

Обработка динамических ситуаций *bounds*, *overflow* и *underflow* в вызывающем модуле выполняется оператором **except**.

3.6. Процедуры

Процедура – один из видов модулей языка Клу, реализующий *абстракцию исполнения*. Процедура может быть описана либо как независимый модуль, либо как операция кластера, либо как *скрытая подпрограмма* кластера, локальная в его реализации. В отличие от других алголоподобных языков, в языке Клу процедуры не могут быть вложенными.

Процедура выполняет некоторые действия над совокупностью *аргументов* и выдает совокупность *результатов*. Кроме того, она может закончиться возникновением нескольких *ситуаций*. Аргументы, типы результатов и имена возможных ситуаций (с типами их аргументов) указываются в *заголовке процедуры*. В общем случае ее описание имеет вид

```
p = proc (a1 : t1, ... , an : tn) returns (r1, ... , rm)
    signals (s1, ... , sk)
    body
end p
```

Здесь p – идентификатор процедуры, a_i – идентификаторы аргументов, t_i – типы аргументов, r_j – типы результатов, s_l – имена ситуаций с типами их возможных аргументов, *body* – тело процедуры. При отсутствии аргументов после слова **proc** указываются пустые скобки ( ), результатов или ситуаций – часть **returns** или **signals** опускается. Возможность возникновения стандартной ситуации *failure* (п. 3.5.6.) по умолчанию подразумевается в любой процедуре (например, из-за использования переменной с неопределенным значением), но указание ее имени в заголовке запрещено.

Каждая процедура имеет свой *тип* (п. 3.3.8), который получается из ее заголовка удалением идентификаторов аргументов a_i и заменой слова **proc** словом **proctype**:

```
proctype (t1, ... , tn) returns (r1, ... , rm) signals (s1, ... , sk)
```

Тип процедуры в языке Клу служит ее спецификацией, полностью определяет ее абстрактное (видимое) поведение и служит в системе Клу для контроля типов при *вызове процедуры:*

$p(x_1, \dots, x_n)$

При вызове число *фактических аргументов* x_i должно быть равно числу формальных аргументов a_i, а их типы должны совпадать с типами t_i, за исключением случая, если t_i – тип **any** (при этом тип аргумента x_i может быть *любым*). Передача аргументов при вызове трактуется как описание локальных переменных a_i с инициализацией их объектами-аргументами:

$a_i : t_i := x_i$

Процедура взаимодействует с другими модулями только через аргументы, результаты и ситуации. Она может использовать также идентификаторы глобальных констант, типов и модулей, входящие в ее контекст компиляции (п. 3.11). Описаний глобальных переменных в языке Клу нет. Если глобальный идентификатор, например *complex*, использован в процедуре *p* в его глобальном смысле (для обозначения кластера), то он не может быть описан в процедуре *p* как локальный идентификатор.

Вызов процедуры может использоваться в следующих конструкциях:

– в качестве оператора, если процедура не выдает результатов либо если они игнорируются;

– в выражении, если процедура выдает ровно один результат;

– в операторе группового присваивания:

$v_1, \dots, v_m := p(x_1, \dots, x_n)$

если процедура *p* выдает *m* результатов ($m > 1$). Последний вариант является единственным в языке Клу способом использования процедуры, выдающей более одного результата.

Завершение процедуры и выдача ее результатов осуществляется оператором

return $(y_1, \dots, y_m)$

где y_j – выражения, число которых должно быть равно числу типов результатов r_j в заголовке процедуры, а тип каждого объекта должен совпадать с типом r_j. Оператор

return

означает завершение процедуры *без* выдачи результатов (в этом случае часть **returns** в заголовке процедуры должна отсутствовать). Тот же эффект вызывает завершение последнего оператора перед заключительным **end**.

Собственные переменные. Процедура *p* может иметь собственные переменные, которые создаются при первом вызове процедуры и сохраняются от вызова к вызову. Например, в процедуре можно описать собственную переменную – счетчик числа ее вызовов:

```
p = proc ( )
    own callnmb: int := 0; . . .
    callnmb := callnmb + 1; . . .
    end p
```

При первом вызове процедуры *p* исполняется описание переменной *callnmb*, которая инициализируется нулем. Далее при каждом новом вызове *p* (в том числе и при первом) значение переменной *callnmb* , увеличивается на единицу.

Как видно из этого примера, собственные переменные удобны для хранения истории вычислений. Их можно также использовать для обозначения "вычисляемых констант" – объектов, которые имеют постоянный смысл, но не являются константами с точки зрения языка Клу и, следовательно, не могут быть описаны с помощью тождества (п. 3.3.8). Например, таблица

own *tab*: **array** [**string**] : = **array** [**string**] $ *new* () (1)

будет доступна в процедуре, в которой локализовано ее описание, причем ее создание выполняется один раз, при первом вызове этой процедуры.

Понятие собственной переменной в языке Клу близко аналогичному понятию в языке Алгол-60.

3.7. Итераторы

Итератор – это модуль, предназначенный для вычисления последовательности объектов, являющихся компонентами некоторого абстрактного объекта сложной структуры (например, множества). Итератор в языке Клу иначе именуется как "абстракция управления", так как он абстрагирует программиста от способа вычисления (перебора) объектов-*компонент* и используется для управления *циклами*, в которых эти компоненты обрабатываются. Итератор может выполнять и более простые повторные вычисления объектов – например, генерацию последовательности целых значений с заданными границами и шагом. Клу – первый язык программирования, в котором эта важная концепция, уже долгое время неявно использующаяся в практике программирования, оформлена в виде независимой языковой конструкции.

Описание итератора и обозначение его типа по форме сходны с описанием и обозначением типа процедуры:

it = **iter** $(a_1 : t_1, \ldots, a_n : t_n)$ **yields** $(r_1, \ldots, r_m)$ **signals** $(s_1, \ldots, s_k)$

body
end *it*

Здесь it – идентификатор итератора, a_i, t_i – идентификаторы и типы аргументов, r_j – типы объектов, выдаваемых итератором, s_l – имена ситуаций с возможными списками типов их аргументов, *body* – тело итератора. В тело могут входить описания собственных переменных (п. 3.6).

В общем случае итератор выдает последовательность кортежей объектов $\{(y_1, \ldots, y_m)\}$, где y_j имеет тип r_j. Тип итератора задается обозначением типа

itertype $(t_1, \ldots, t_n)$ **yields** $(r_1, \ldots, r_m)$ **signals** $(s_1, \ldots, s_k)$

Итератор, как и процедура, может быть *параметризован* (п. 3.9).

Принцип работы итератора следующий. Он запускается при исполнении заголовка цикла:

for $i_1 : r_1, \ldots, i_m : r_m$ **in** $it\,(x_1, \ldots, x_n)$ **do** *body* **end**

Переменные i_j играют роль параметров цикла. Итератор, запускаемый при очередном повторении цикла, присваивает переменным i_j ссылки на объекты $y_1, \ldots, y_m$, выдаваемые оператором

yield $(y_1, \ldots, y_m)$

После этого итератор передает управление телу цикла *body*. При следующем повторении цикла итератор продолжает работу, начиная с оператора, динамически следующего за оператором **yield**, до выполнения такого же оператора, выдающего новые ссылки на объекты. При этом значения локальных

переменных итератора сохраняются, и их можно использовать для генерации новых объектов y_j.

Итератор завершается в результате выполнения оператора **return** (результаты не допускаются) или последнего внутреннего оператора. При этом завершается и управляемый им цикл.

Программисту рекомендуется [83] при организации циклов придерживаться следующих принципов:

– тело цикла не должно модифицировать объекты-аргументы итератора; это повышает надежность программы (обобщение известного принципа, реализованного в большинстве языков: в теле цикла не должны модифицироваться границы его изменения индексов);

– итератор, как правило, не должен модифицировать свои аргументы (возможны исключения, связанные с хранением "текущего указателя" внутри анализируемого объекта).

П р и м е р.

Работа с текстом: выделение и обработка слов. Текст задается в виде строки и состоит из слов, разделенных пробелом.

```
words = iter (s : string) yields (string)
   space = ' ';
   cur : string := s;
   size : int := string $ size (cur);
   k : int;
   while size ~ = 0 do
      k := string $ indexc (space, cur);
      if k = 0
      then yield (cur);
           size := 0
      else
           yield (string $ substr (cur, 1, k – 1));
           if k = size
           then size := 0
           else cur := string $ substr (cur, k + 1, size – k);
                size := size – k
           end
      end
   end
  end words;
  . . .
  text : string := "humpty dumpty sat on a wall";
  for word : string in words (text) do
     writestring (word);
     writestring ("size=");
     writeint (string $ size (word));
     writeln
  end
```

Другие примеры итераторов приведены в пп. 3.2 и 3.8, примеры их использования в циклах – в пп. 3.2, 3.3.2, 3.3.7, 3.5.4, 3.5.6. Итераторы входят в состав операций для встроенных типов **int** и **string**, встроенных генераторов типов **array** и **sequence**.

§ 3.8. Кластеры

Кластер – это форма описания АТД в языке Клу. Описание кластера определяет новый тип данных, отличных от всех других типов. Описание параметризованного кластера (п. 3.9) определяет класс родственных АТД, отличающихся значениями параметров. Встроенные типы и генераторы типов считаются соответственно кластерами и параметризованными кластерами. Описание кластера имеет вид

$c =$ **cluster is** $p_1, \ldots, p_n$
  rep $= t$;
  $p_1 =$ **proc** $(\ldots) \ldots$ **end** $p_1; \ldots$
  $p_n =$ **proc** $(\ldots) \ldots$ **end** p_n
end c

где c – идентификатор кластера; **rep** $= t$ – описание *конкретного представления* объектов типа c, означающее, что внутри кластера они могут рассматриваться как объекты типа t; $p_1, \ldots, p_n$ – идентификаторы *операций*, осуществляющих обработку объектов типа c. Операции p_i могут быть как *процедурами*, так и *итераторами*.

В описание кластера могут входить *тождества*, вводящие обозначения для типов и констант, локальные внутри кластера; описания *собственных переменных* кластера и описания *скрытых подпрограмм*, используемых при реализации операций p_i. Все эти описания локализованы в кластере и недоступны извне.

Операции p_i определяют *абстрактное поведение* объекта типа c, т.е. элементы его *состояния*, доступные программисту для анализа и модификации. Вне кластера объекты типа c считаются абстрактными и доступны только через операции p_i. Конкретное представление и реализация операций скрыты внутри кластера. Они выражаются через встроенные или уже определенные кластеры. В конечном счете исполнение программы на языке Клу, как и на любом другом языке, выражается в терминах реализации встроенных типов, однако "видимым" для программиста является лишь самый высокий уровень абстракции.

Для типа конкретного представления t в теле кластера удобно использовать стандартное обозначение **rep**. Обозначение типа **cvt**, задающее тип аргумента или результата операции p_i, означает, что в теле операции объект-аргумент типа c рассматривается как объект типа **rep**, а вне ее (при передаче аргумента и выдаче результата) – как объект типа c. Имеются и явные операции преобразования типов $c \leftrightarrow$ **rep**, которые используются в реализации операций p_i: **down** (x) – преобразование типа c к типу **rep**; **up**(r) – преобразование типа **rep** к типу c.

Вызов операции p из кластера c выполняется по ее составному имени $c \$ p$:

$c \$ p(x_1, \ldots, x_k)$,

где x_i – объекты-аргументы. При вызове необходимо соблюдать *интерфейс* операции, определенный ее типом (пп. 3.6, 3.7): число и типы аргументов x_i должны соответствовать типу операции; в вызывающем модуле необходимо предусмотреть использование результатов операции p и обработку ее ситуаций в соответствии с ее типом.

С о с т а в   о п е р а ц и й   к л а с т е р а. В языке Клу нет формальных требований к набору операций $p_1, \ldots, p_n$ кластера *c*: они могут быть любыми. Однако для удобства и с целью унификации работы с объектами определяемых и встроенных типов программисту рекомендуется включать в определение любого кластера *c* операции:

1. *create*: **proctype** (...) **returns** (*c*) – создание абстрактных объектов типа *c* (генератор объектов);

2. *equal*: **proctype** (*c, c*) **returns** (**bool**) – сравнение двух объектов типа *c* на равенство. Равенство объектов понимается по-разному, в зависимости от предполагаемого поведения объекта. Если объекты типа *c постоянны*, то осуществляется проверка на совпадение структуры и поэлементное равенство (например, два комплексных числа равны, если попарно равны их вещественные и мнимые части). При этом требуется, чтобы тип элементов (компонент объектов типа *c*) содержал операцию равенства (*equal*). Если кластер *c* описывает *изменяемые* объекты, то равенство понимается как совпадение объектов. Например, для встроенного кластера **array** [*t*] результат операции *equal* истинен в том и только в том случае, если аргументы ссылаются на один и тот же массив. Операция *equal* имеет инфиксную форму: $x = y$.

3. *similar*: **proctype** (*c, c*) **returns** (**bool**) – сравнение двух объектов типа *c* на сходство их состояний. В отличие от операции *equal* операция *similar* имеет одинаковый смысл и для постоянных, и для изменяемых объектов: результат операции истинен, если состояния ее аргументов сходны; например, оба являются последовательностями с одним и тем же числом элементов, причем элементы попарно сходны (т.е. над ними выполняются отношение *similar*). Применение операции *similar* требует наличия этой же операции у типа элементов (компонент) объекта.

4. *copy*: **proctype** (*c*) **returns** (*c*) – полное копирование объекта *x* типа *c*, т.е. создание нового объекта *y* типа *c*, структура которого сходна со структурой исходного объекта, а элементы *y* являются полными копиями элементов *x*. Применение операции *copy* требует наличия у типа элементов операции *copy*. Объект, получаемый в результате операции *copy*, и исходный объект не должны быть связаны между собой (изменение состояния одного из них не должно вызывать изменения состояния другого).

Операции *create*, *equal*, *similar* и *copy* составляют рекомендуемый минимальный набор операций кластера, определяющего тип постоянных объектов. Для изменяемых объектов, кроме перечисленных, рекомендуются следующие операции.

5. *similar*1 : **proctype** (*c, c*) **returns** (**bool**) – сравнение двух объектов, выполняемое так же, как операция *similar*, но использующее операцию *equal* для сравнения элементов. Соответственно для применения этой операции тип элементов *c* должен содержать операцию *equal*.

6. *copy*1 : **proctype** (*c*) **returns** (*c*) – частичное копирование объекта типа *c*, т.е. создание нового объекта с таким же состоянием, как у исходного объекта. Например, результатом операции *copy*1 для массива является такой же массив, элементы которого ссылаются на *те же самые* объекты, что и элементы исходного массива.

Между результатами операций *equal*, *similar* и *copy* должны выполняться следующие очевидные логические соотношения [83]:

$$\forall x, y \in c \quad c\$equal(x, y) \supset c\$similar(x, y)$$

$$\forall x \in c \quad c\$similar(x, c\$copy(x))$$

Префиксная и инфиксная форма операций. Ряд операций, определенных над встроенными типами и доступных через принятые в языке Клу составные имена и префиксную форму, имеют уже сложившиеся обозначения и традиционную инфиксную форму в большинстве языков программирования. С целью сохранения привычной структуры программы в языке Клу для этих операций введена в качестве сокращения инфиксная форма записи. При этом однозначно зафиксирована связь между символьным *именем* операции (в процедурной форме) и ее *знаком* (в инфиксной форме). Приведем список этих операций. Обозначения: x, y, z – аргументы операций; f – имя поля; t – тип аргумента x. Слева – полная форма, справа – сокращенная инфиксная.

Арифметические операции:

t$add(x, y)	*x + y*
t$sub(x, y)	*x – y*
t$mul(x, y)	*x * y*
t$div(x, y)	*x/y*
t$mod(x, y)	*x//y*
t$power(x, y)	*x ** y*
t$minus(x)	*– x*

Операции отношения:

t$equal(x, y)	*x = y*
t$lt(x, y)	*x < y*
t$le(x, y)	*x <= y*
t$ge(x, y)	*x >= y*
t$gt(x, y)	*x > y*

Логические операции:

t$and(x, y)	*x & y*
t$or(x, y)	*x | y*
t$not(x)	*~x*

Связь между операциями отношения и логическими операциями:

~(x < y)	*x ~< y*
~(x <= y)	*x ~<= y*
~(x >= y)	*x ~>= y*
~(x > y)	*x ~> y*
~(x = y)	*x ~= y*

Операции над последовательностями:

t$concat(x, y)	*x ‖ y*

Операции над массивами:

t ∮ *fetch* (x, y)	$x[y]$
t ∮ *store* (x, y, z)	$x[y] := z$

Операции над записями:

t ∮ *get_f* (x)	$x.f$
t ∮ *set_f* (x, y)	$x.f := y$

Перечисленные операции используются для встроенных типов и имеют определенную семантику, которая, как и знак операции, ассоциируется с ее символьным именем. Сокращенные обозначения операций разрешается использовать и в том случае, если t – определяемый тип, содержащий операции с теми же именами и сходным интерфейсом. Например, если t – кластер "матрица", содержащий определение операции *add* с интерфейсом:

proctype (t, t) **returns** (t),

то вместо t ∮ *add* $(m1, m2)$ можно использовать форму $m1 + m2$. Такое расширение обеспечивает как удобство записи операций, так и их *полиморфизм* (смысл операции $x + y$ определяется типом первого аргумента). Рекомендуется, чтобы при определении кластера с операциями *add*, *sub*, *equal* и т.д. сохранялась и их семантика, подобная семантике соответствующих встроенных операций и, таким образом, сохранялось бы и поведение объекта, подобное поведение объектов встроенных типов. Например, кластер с операциями *fetch* и *store* должен определять тип изменяемых объектов, подобных массивам по своему поведению (см. пример описания кластера-таблицы в п. 3.9). Разумеется, выполнение этих семантических рекомендаций не контролируется клу-системой и является лишь элементом дисциплины программирования; язык и система Клу обеспечивают чисто формальную связь имени и знака операции.

Полнота или минимальность набора операций. При проектировании набора операций некоторого кластера один из основных вопросов состоит в том, следует ли стремиться к минимальности числа операций и их семантической ортогональности либо к его полноте и удобству использования. Из практических соображений технологии разработки больших программ критерий полноты и удобства представляется более предпочтительным. Минимальность и ортогональность набора операций – свойства, важные в основном с формально-алгебраической точки зрения. Например, набор встроенных операций над массивами включает операции *low*, *high*, *fetch* и *store*, достаточные для обработки массивов в цикле в стиле языков Алгол и Фортран. Однако применение операций-итераторов *elements* и *indexes* позволяет организовать обработку элементов массива более лаконичными и удобными средствами (п. 3.3.2).

Пример.

Кластер "двоичное дерево".

```
b_tree = cluster is create, is_empty, get_left, get_right,
get_inf, set_node, nodes, equal, similar, similar1, copy, copy1
    node = record [inf : int, left, right : b_tree];
    rep = variant [empty : null, node : node];
```

```
create = proc (    ) returns (cvt)
          return (rep $ make_empty (nil))
          end create;
is_empty = proc (t : cvt) returns (bool)
               return (rep $ is_empty (t))
              end is_empty;
get_left = proc (t : cvt) returns (b_tree) signals (empty)
               tagcage t
                 tag node (n : node): return (n.left)
                 tag empty : signal empty
               end
            end get_left
get_right = proc (t : cvt) returns (b_tree) signals(empty)
              tagcase t
               tag node (n : node): return (n. right)
                tag   empty : singnal empty
              end
             end get_right
get_inf = proc (t : cvt) returns (int) signals (empty)
             tagcase t
             tag node (n : node) : return (n. inf )
             tag empty : signal empty
             end
           end get_inf
set_node = proc (t : cvt, i : int, l, r : b_tree)
               tagcase t
                 tag node (n : node) :
                      n. inf := i ; n. left := l; n. right := r
                 tag empty :
                    rep $ change_node
                           (t, node $ { inf : i, left : l; right : r } )
                    % изменение признака и значения  t
               end
             end set_node
nodes =  iter (t : cvt) yields (int)
           tagcase t
              tag node (n : node) :
                   yield (n.inf );
                   for i : int in nodes (n. left) do
                        yield (i)
                   end;
                   for k : int in nodes (n. right) do
                      yield (k)
                    end
              tag empty :
            end
          end nodes;
equal  = proc (t1, t2: cvt) returns (bool)
                return (t1 = t2)
```

```
        end equal;
    similar = proc (t1, t2: cvt) returns (bool)
      tagcase t1
        tag empty:
            return (rep $ is_empty (t2))
         tag node (n1 : node)):
           tagcase t2
             tag empty : return (false)
             tag node (n2: node) :
               if n1. inf ~ = n2.inf
                then return (false)
                else return
                     (similar (n1.left, n2. left) cand
                     similar(n1.right, n2. right))
                end
            end % t2
       end % t1
     end similar;
    similar1 = proc (t1, t2 : cvt) returns (bool)
      tagcase t1
        tag empty :
            return ( is_ empty (t2))
         tag node (n1 : node):
          tagcase t2
            tag empty: return (false)
            tag node (n2: node):
           return (n1 = n2)
           end % t2
     end % t1
    end similar1;
    copy = proc (t : cvt) returns (cvt)
             tagcase t
               tag empty: return (rep $ make_empty (nil))
               tag node (n : node);
                    return
                      (rep $ make_node
                         (node $ { inf : n. inf,
                              left : copy (n.left),
                              right : copy (n.right)} ))
             end
    end copy
    copy1 = proc (t : cvt) returns (cvt)
               tagcase t
                  tag empty :  return (rep $ make_empty (nil))
                    tag node (n : node):
                       return (rep $ make_node (n))
               end
    end copy1
end b_tree
```

Кластер *b_tree* определяет АТД "полное двоичное дерево". Узел дерева содержит информацию в виде целого числа. Рекурсивность типа *b_tree* выражается через его конкретное представление. Деревья считаются изменяемыми объектами. Построение и модификация дерева выполняются операциями *create* (создание пустого дерева) и *set_node* (создание или изменение узла). Операции *get_left, get_right* и *get_inf* обеспечивают доступ к элементам корня непустого дерева *t* в наиболее удобной форме: *t.left, t.right* и *t.inf.* Если дерево *t* пустое, возникает ситуация *empty.* Итератор *nodes* выдает значения информационных элементов в порядке прямого левостороннего обхода. Операция *is_empty* выполняет проверку на пустое дерево. Операция *get_left, get_right* и *is_empty* позволяет осуществить анализ узлов дерева. Операции *equal, similar, similar*1, *copy* и *copy*1 имеют обычный для изменяемых объектов смысл.

Итератор *nodes* и процедура *similar* рекурсивны. Для рекурсивного вызова итератора *nodes* в его теле организуется цикл по левому и по правому поддереву аргумента-дерева *t*.

Пример использования кластера *b_tree*:

```
t : b_tree := b_tree $ create ( )
t1: b_tree := b_tree $ create ( );
t2: b_tree := b_tree $ create ( );
b_tree $ set_node (t, 0, t1, t2);
b_tree $ set_node (t1, 1, b_tree $ create ( ), b_tree $ create ( ));
b_tree $ set_node (t2, 2, b_tree $ create ( ), b_tree $ create ( ));
for i : int in b_tree $ nodes (t) do
    writeint (i)
end % output : 0 1 2
```

§ 3.9. Параметризованные модули

Модули всех трех видов – процедуры, кластеры и итераторы – могут быть *параметризованными.* С помощью параметризованного модуля определяется класс родственных модулей, отличающихся значениями параметров. Описания формальных параметров задаются в заголовке параметризованного модуля после слов **proc**, **iter** или **cluster** в форме

$[x_1 : t_1, \ldots, x_n : t_n]$,

где x_i – идентификаторы параметров, t_i – типы параметров. Параметры x_i могут быть либо *константами* типа **int**, **real**, **bool**, **char**, **string** или **null**, либо *типами* (в этом случае тип параметра t_i специфицируется словом **type**).

Параметр-тип – наиболее важный вид параметризации, обеспечивающий программирование *полиморфных операций*. Идентификатор параметра-типа, специфицированного как t: **type**, может использоваться в любом месте модуля в качестве обозначения типа, идентификаторы параметров-констант – в качестве выражений.

Для использования параметризованного модуля *m* необходима его *конкретизация* $m[y_1, \ldots, y_n]$, где *m* – идентификатор модуля; y_i – значения *фактических параметров.* Результатом конкретизации параметризованной процедуры или итератора являются соответственно процедура или итератор, параметризованного кластера – кластер (тип). Типы параметров-

констант y_i должны совпадать с типами t_i формальных параметров x_i; если t_i – спецификация type, то фактический параметр y_i должен быть обозначением типа.

Особый случай представляет использование операций параметра-типа t в параметризованном модуле m. В этом случае необходимо, чтобы фактический параметр-тип имел операции с заданными именами и с интерфейсом, соответствующих их использованию. *Ограничения* на состав и интерфейс операций фактических параметров-типов указываются в конце заголовка параметризованного модуля в форме

where t **has** $p_1 : pt_1, \ldots, p_k : pt_k$

где t – формальный параметр-тип; p_i – имена операций, наличие которых требуется от фактического параметра-типа; pt_i – типы этих операций (**proctype** или **itertype**). В теле модуля могут использоваться только те операции параметра-типа t, которые указываются в ограничениях:

$t \$ p_i(x_1, \ldots, x_m)$

где аргументы x_i соответствуют типу операции pt_i. Контроль соответствия фактического параметра-типа ограничениям выполняется при конкретизации.

В качестве примера рассмотрим описание кластера *b_tree* (п. 3.8). Чтобы преобразовать его описание в описание параметризованного кластера (с произвольным типом информации в узлах дерева), достаточно после слова **cluster** поместить спецификацию параметра-типа [*inf_type* : **type**] и везде **int** заменить на *inf_type*.

Пример.

Параметризованная полиморфная процедура сортировки массива.

```
sort = proc[t : type] (a : array [t])
      where t has lt : proctype (t, t) returns (bool)
      at  = array [t] ;
      l : int := at $ low (a);
      h : int := at $ high (a); x : t; k : int;
      for i : int in int $ from_to (l, h – 1) do
          x := a[i] ;
          for j : int in int $ from_to (i + 1, h) do
              if a[j] < x then x := a[j] ; k := j end end;
          a[i], a[k] := x, a[i]
      end
      end sort
```

Алгоритм процедуры *sort* применим к массивам из элементов любого типа, содержащего двуместную операцию *lt* с логическим результатом: **int**, **real**, **bool**, **char**; определяемых типов, в которых описана такая операция. Процедура сортировки для конкретного типа получается из процедуры *sort* путем конкретизации параметра-типа t:

```
int_sort = sort[int] ;
string_sort = sort[string]
```

Если ограничения where содержат часто повторяющиеся спецификации операций, то удобно использовать конструкцию "множество типов":

```
ordered_types =
{ t | t has equal, le, lt, ge, gt : proctype (t, t) returns (bool) } ;
p = proc[ s, u, w : type] . . .
        where s in ordered_types, u  in ordered_types,
                w in ordered_types . . .
```

Если кластер c имеет параметр-типа t:

```
c = cluster [t : type]  is p, q, r . . . where t has . . . . . .
```

то в описании операций кластера p, q, r на параметр кластера t могут накладываться *дополнительные ограничения*:

```
p = proc . . . where t has . . .
```

Иначе говоря, некоторые из операций кластера с параметрами могут потребовать от типа-параметра наличия дополнительных свойств. Например, операция *fetch* определена над любыми массивами, а операция *similar* – только над такими, тип элементов которых имеет операцию *similar*. Это условие на языке Клу можно выразить следующим образом:

```
array = cluster [t: type]  is fetch, similar, . . .
            . . .
            similar = proc (a, b  : cvt) returns (bool)
              where t has similar : proctype (t, t) returns (bool)
. . . end array
```

П р и м е р.
Параметризованный кластер "ассоциативная таблица":

```
table = cluster[t : type]  is create, insert, fetch, store,
            size, indexes, elements, equal , similar, similar1, copy, copy1
            elem = record[key  : string, elem  : t] ;
            rep = array  [elem] ;
            create = proc (   ) returns (cvt)
                        return (rep $ new (   ))
                        end create;
            index = proc (a : rep, s :  string) returns (int)
                        % a hidden routine
                        for i : int in rep $ indexes (a) do
                           if a [i] . key = s
                           then return (i)
                           end
                        end;
                        return (0)
                    end index;
              insert = proc (tab : cvt, s : string, x : t)
                        signals (wrong_index)
                        if index (tab, s) > 0
                        then signal wrong_index
                        else
```

```
                    return (rep $ addh (tab, elem $ { key : s, elem : x } ))
                    end
                end insert;
            fetch = proc (tab : cvt, s : string) returns (t) signals (bounds)
                    i : int := index (tab, s);
                    if i = 0 then signal bounds
                    else return (tab [i]. elem)
                    end
                   end  fetch;
            store = proc (tab : cvt, s : string, x : t)  signals (bounds)
                    i : int := index (tab, s);
                    if i = 0 then signal bounds
                    else tab[i] . elem := x
                    end
                   end store;
    size = proc (tab : cvt) returns  (int)
              return (rep $  size (tab))
            end size;
    indexes = iter (tab : cvt) yields (string)
                  for e : elem in rep $  elements (tab) do
                      yield (e. key)
                  end end indexes;
    elements  = iter (tab : cvt) yields (t)
                  for e : elem in rep $  elements (tab) do
                      yield (e. elem)
                  end
                  end elements;
    equal = proc (tab1, tab2 : cvt) returns (bool)
                  return (tab1 = tab2)
              end equal;
    similar = proc (tab1, tab2 : cvt) returns (bool)
                where t has similar : proctype (t, t) returns (bool)
                      if rep $  size (tab1) ~ = rep $  size (tab2)
                      then return (false)
                      else
                      for i : inf in rep $  indexes (tab1) do
                          if ~ rep $  similar (tab1 [i], tab2[i] )
                          then return (false)
                          end
                       end;
                       return (true)
                       end
                end similar;
  similar1 = proc (tab1, tab2 : cvt)  returns (bool)
              , where t has equal : proctype (t, t) returns (bool)
                if rep $  size (tab1) ~= rep $  size (tab2)
                then return (false)
                else for i : int in rep $  indexes (tab1) do
                  if tab1  [i] key ~ = tab2[i] . key
```

```
                then return (false)
                elseif ~ t $ equal (tab1 [i] . elem, tab2 [i] . elem)
                then return (false)
                end end;
              return (true)
              end end similar1;
    copy = proc (tab : cvt) returns (cvt)
              where t has copy : proctype (t) returns (t)
              c : rep := rep $ new ( );
              for e : elem in rep $ elements (tab) do
                  rep $ addh (c, elem $ { key:e. key, elem : t $ copy (e.elem) } )
              end; return (c)
              end copy;
  copy1 = proc (tab : cvt) returns (cvt)
            c : rep := rep $ new ( );
            for e : elem in rep $ elements (tab) do
                rep $ addh (c, elem $ { key : e. key,elem : e.elem } )
            end;
            return (c)
         end copy1
  end table
```

Кластер *table* определяет класс родственных абстракций, выражающих понятие *ассоциативной таблицы*. Таблица – изменяемый объект, состоящий из *элементов* типа-параметра *t*. В качестве *индексов* в таблице используются строки. Операция *fetch*, *store*, *size*, *elements*, *indexes* обеспечивают работу с таблицей в таких же терминах и обозначениях, как с массивом. Операция *insert* добавляет в таблицу новый элемент. Состояние таблицы в реализации поддерживается таким образом, что повторение значений индексов-строк не допускается. При попытке добавления элемента с повторяющимся значением индекса возникает ситуация *wrong_index*. В теле кластера описана скрытая процедура *index*, устанавливающая соответствие между абстрактными индексами-строками и индексами массива – конкретного представления. Описание кластера не накладывает никаких ограничений на тип-параметр *t*, однако операции *similar*, *similar*1 и *copy* содержат дополнительные ограничения на тип *t* (для каждой операции – свои).

Пример.

Использование кластера *table*:

```
tr = table[real] ;
t : tr := tr $ create ( );
tr $ insert (t, "small", 1e – 10);
tr $ insert (t, "large", 1e20);
writereal (t ["small"] );
t ["large"] : = 1e30
```

3.10. Библиотека модулей

Ядро языка Клу, описанное в пп. 3.1–3.9, спроектировано таким образом, что все модули (процедуры, итераторы и кластеры), составляющие клу-программу, могут отдельно компилироваться и параллельно разрабатываться. Для того чтобы использовать модуль *m*, достаточно знать его интерфейс; реализация модуля *m* может разрабатываться одновременно с реализацией модулей, использующих модуль *m*. В этом и состоит принятый в языке Клу подход к пошаговой разработке программ. Реализация языка Клу должна обеспечить хранение информации об интерфейсах модулей в некоторой базе данных и использование этой информации для контроля типов и связей между модулями. Напомним, что интерфейс процедуры или итератора – это его тип (п. 3.3.8), а интерфейс кластера есть совокупность интерфейсов его операций. В интерфейс параметризованного модуля включаются также типы параметров.

Другая проблема, возникающая в языке Клу, – это сборка клу-программы для исполнения. В Клу нет явного описания программы, как, например, в языке Паскаль. Клу-программа образуется из тех модулей и их конкретных параметризаций, которые прямо или косвенно вызываются в результате вызова некоторой *головной процедуры* с конкретными значениями аргументов. Таким образом, исполнению программы должен предшествовать процесс ее *сборки,* при котором определяется совокупность модулей, составляющих программу, и выполняются их необходимые конкретизации. При сборке требуется полная информация о модулях, их интерфейсах и реализациях.

Наконец, третья важная проблема, решение которой в описании языка Клу [83] не изложено с полной ясностью, – это проблема глобальных идентификаторов. Как уже отмечалось, в языке Клу нет явно выраженных глобальных описаний (в частности, глобальные переменные запрещены). Однако для связи модулей и для удобства программирования необходим способ именования модулей, констант и обозначений типов. Для этого в описание языка Клу вводится понятие *контекста компиляции.* Ему не дано строгого определения, но, как видно из различных фрагментов описания языка [83], его подразумеваемый смысл следующий. Контекст компиляции модуля – это совокупность тождеств (п. 3.4.3), вводящих идентификаторы для констант, обозначений типов и других модулей, используемых в некотором модуле. Контекст компиляции передается транслятору вместе с текстом модуля, использующего глобальные обозначения из этого контекста. В большой программной системе модуль может иметь несколько (возможно, перекрывающихся) контекстов компиляции, каждый из которых соответствует всему проекту или его части. Способ указания ссылок на модули в правых частях тождеств, образующих контекст компиляции, зависит от реализации клу-системы.

Таким образом, в описании языка Клу ключевые вопросы контроля связей между модулями, пошаговой разработки программы, ее сборки и организации контекста глобальных идентификаторов вынесены за пределы языка, а их решение должно быть принято при реализации. В описании языка нет даже явного определения синтаксической структуры раздельных описаний интерфейса, реализации и контекста компиляции как неза-

висимых единиц трансляции. Для хранения информации о модулях, поддержки пошаговой разработки программ и контроля связей между модулями в реализации Клу авторы языка рекомендуют использовать специализированную базу данных – *библиотеку модулей.* По мнению авторов Клу, библиотека должна иметь иерархическую древовидную структуру; в узлах дерева должна содержаться информация о модулях. При этом горизонтальный слой вершин дерева соответствует традиционному понятию "уровень абстракции".

Практика программирования, однако, показывает, что дисциплина проек ирования и разработки программ, при которой в качестве основы используется дерево модулей, для больших задач неприемлема: требуются более сложные сетевые связи модулей. Например, один и тот же кластер "матрица" может быть использован в кластере "линейная алгебра" и в кластере "цепи Маркова", возможно, определенных на разных уровнях абстракции. Необходимость реализации такого рода связей между модулями нарушает принцип древовидности структуры библиотеки и превращает ее в ациклический ориентированный граф. По-видимому, наиболее практически удобна такая структура библиотеки модулей, которая обеспечивает доступ из любой вершины графа библиотеки к любой другой. При этом система Клу не должна допускать циклических зависимостей в определениях кластеров и других модулей. В п. 3.11.2 рассматривается организация библиотеки модулей в системе Клу-Эльбрус, удовлетворяющая этим принципам.

3.11. Система программирования Клу-Эльбрус

Система программирования Клу-Эльбрус [18] предназначена для проектирования, разработки, отладки и сопровождения больших программных комплексов, написанных на языке Клу. Система Клу-Эльбрус является первой отечественной реализацией языка Клу и одной из перых систем поддержки технологии программирования с АТД. В силу особенностей базового языка Клу и технологических свойств самой системы программирования, система Клу-Эльбрус способствует разработке модульных, наглядных и надежных программ.

Система Клу-Эльбрус состоит из следующих компонент:

– компилятор с языка Клу;

– система управления библиотекой модулей:

– система динамической поддержки.

Система выполняет следующие основные функции:

– раздельная компиляция программных модулей на языке Клу;

– поддержка пошаговой разработки программ на языке Клу с использованием библиотеки модулей;

– статическая сборка программ на языке Клу для последующего исполнения;

– поддержка управления модулями, ввода-вывода и диагностики ошибок при исполнении клу-программ.

В системе Клу-Эльбрус реализовано два режима сборки программы – *статический и динамический.* Статическая сборка обеспечивает получение более эффективных программ и рекомендуется для их отлаженных версий,

предназначенных для многократного использования. Динамическая сборка предназначена для автономной отладки отдельных модулей в составе уже отлаженного комплекса или отладочной программы-драйвера.

Предусмотрена также оптимизация обращений к абстрактным операциям кластеров: по специальному прагмату они транслируются как открытые подстановки в место их вызова.

Эти свойства системы Клу-Эльбрус в сочетании с полнотой реализации языка Клу позволяют считать ее производственной системой программирования, предназначенной для создания реальных программ с использованием всего набора языковых средств, составляющих концепцию АТД в ее современном практическом понимании (п. 3.1).

3.11.1. Компиляция. В системе Клу-Эльбрус имеются следующие единицы компиляции:

– полное описание независимого модуля (процедуры, кластера, итератора);

– описание интерфейса модуля;

– описание реализации модуля;

– описание контекста компиляции.

Описания интерфейса и реализации. В языке Клу [83] нет возможности раздельного описания интерфейса модуля и описания его реализации. Однако для пошаговой разработки программ такая возможность необходима. Поэтому во входном языке системы Клу-Эльбрус введены раздельные описания интерфейса и реализации.

П р о ц е д у р а и и т е р а т о р. Согласно описанию языка [83] интерфейс процедуры или итератора определяется его *типом*. Поэтому описание интерфейса имеет вид:

p = proc spec
    proctype (t_1,. . . , t_n) returns (r_1,. . . , r_m) signals (s_1,. . . , s_k)
    end p;
i = iter spec
    itertype (t_1,. . . , t_n) yields (r_1 .. . . , r_m) signals (s_1,. . . , s_k)
    end i

В описании используется сокращение spec (от термина *specification* – спецификация; полное название – *interface specification*, спецификация интерфейса).

Если модуль параметризованный, то в интерфейс включаются также его параметры и ограничения:

pp = proc [x_1 : t_1, . . . , x_p : t_p] spec where . . .
    proctype . . .
    end pp

Описание реализации процедуры (итератора) с заданным интерфейсом не содержит списка параметров, типов результатов, аргументов и имен ситуаций, так как все это уже известно из интерфейса:

p = proc body (a_1, . . . , a_n)
    % тело процедуры
    end p

Здесь a_i – идентификаторы аргументов.

К л а с т е р. Описание интерфейса кластера состоит из его идентификатора, возможных параметров и интерфейсов его операций:

c = **cluster** [x_1 : t_1, . . . , x_p : t_p] **spec is** p_1, . . . , p_n
    p_1 = **proc spec** . . . **end** p_1;
    . . .
    p_n = **iter spec** . . . **end** p_n
end c

Описание реализации кластера с уже известным интерфейсом имеет вид:

c= **cluster body**
    rep = t;
    p_1= **proc body** . . . **end** p_1;
    . . .
    p_n = **iter body** . . . **end** p_n
end c

Кроме конкретного представления и реализации операций p_i в описание реализации кластера c могут входить также тождества и собственные переменные (п. 3.8).

Описания интерфейса и *реализации* могут компилироваться раздельно, однако компиляция интерфейса должна предшествовать компиляции реализации. Описание реализации модуля, использующего модуль t с известным интерфейсом, может компилироваться до описания реализации модуля t.

Описание контекста компиляции. Контекст компиляции в системе Клу-Эльбрус – это последовательность тождеств, связывающих идентификаторы с константами, обозначениями типов и ссылками на модули. Ссылка на модуль задается внешним именем файла (п. 1.23.3), содержащего объектный код описания интерфейса, реализации или полного описания модуля.

П р и м е р описания контекста компиляции:

ar = **array** [**real**];
eps = 1*e* – 15;
pi_div_2 = 1.57;
vector = *ксн*//*клу*//*биб*//*вект*;
matrix = *ксн*//*клу*//*биб*//*матр*

Контекст компиляции может быть задан в текстовом виде перед текстом описания модуля, передаваемым компилятору. Он может транслироваться и как отдельная единица компиляции, а затем передаваться транслятору в качестве параметра при компиляции модуля. Установка контекста компиляции может также выполняться директивами управления библиотекой (п. 3.11.2).

Лексика. В соответствии с описанием языка Клу во входном языке системы Клу-Эльбрус служебные слова (например, **cluster**) не выделяются какими-либо символами и являются зарезервированными идентификаторами. К алфавиту языка добавлены русские буквы; введена русская лексика (см. приложение 4). Вместо символа ∮ используется символ ¤, вместо фи-

гурных скобок – комбинация символов [* и *]; вместо символа ~ – символ ¬. Однако в примерах данной главы использована эталонная лексика языка Клу.

Обращение к компилятору. Компилятор языка Клу – это программа-процедура (п. 1.18.2), форма обращения к которой в максимально возможной степени унифицирована с обращением к компиляторам Эль-76 и Паскаль-Эльбрус. Пусть Клу – имя программы транслятора. Тогда обращение к нему имеет вид

клу (*текст, бибкон, код, рфок, спр, фссылок, терм*)

Смысл параметров несколько отличается от общепринятого в системе "Эльбрус":

1) *текст* – ссылка на файл входного текста, содержащий текст единицы компиляции;

2) *бибкон* – ссылка на библиотеку модулей (п. 3.11.2) или контекст компиляции. Если в исходном тексте используются директивы управления библиотекой модулей, то вторым параметром должна быть ссылка на библиотеку (справочник). При этом контекст компиляции модуля может быть задан либо непосредственно перед текстом модуля, либо с помощью директивы управления библиотекой. Если единица компиляции – описание интерфейса или реализации модуля, то параметром *бибкон* может быть ссылка на файл объектного кода контекста компиляции. Наконец, параметр *бибкон* может быть равен нулю; при этом третьим параметром должна быть передана ссылка на файл для объектного кода модуля:

3) *код* – ссылка на файл для объектного кода единицы компилляции. Если используются директивы управления библиотекой, то параметр *код* может быть равен нулю. В этом случае внешнее имя файла объектного кода определяется этими директивами;

4) *рфок* – контейнер для дополнения к файлу объектного кода. По умолчанию (при *рфок* = 0) ДФОК создается в том же контейнере, что и файл объектного кода единицы компиляции;

5) *спр* – справочник – основа внешних имен библиотечных карт (¤ *биб*), задающих разбиение исходного текста на части; по умолчанию в диалоге – текущий контекст *кн*;

6) *фссылок* – файл для генерации файла ссылок в режиме ¤ *уст словарь* (аналогично паскаль-транслятору, см. п. 2.4.1);

7) *терм* – ссылка на терминальный файл для выдачи протокола (если входной текст задается не с терминала).

Результат процедуры-функции *клу* – число ошибок в тексте единицы компиляции.

В диалоговом режиме компилятор Клу может вызываться директивой стандартного диалога *тр* (п. 1.33.8), в форме

тр//текст и2//бибкон : //клу к //код > кcн//рфок
(//спр) сл//фссылок

где *//текст, //бибкон, //код, ксн//рфок, //спр, //фссылок* – внешние имена, задающие параметры компилятора; *//клу* – внешнее имя его файла объектного кода (указан наиболее полный вариант директивы).

Режимы и сообщения компилятора в основном аналогичны по форме режимам и сообщениям компилятора Паскаль-Эльбрус (п. 2.4.2, 2.4.8). Например, управляющая карта

¤ *уст л к а си* ¤ *максош* = 100

устанавливает следующие режимы транслятора:

к – распечатка объектного кода;

л – распечатка исходного текста;

а – распечатка адресных пар (относительных адресов) аргументов и переменных;

си – распечатка информации о сегментации;

¤ – распечатка управляющих карт;

максош = 100 – увеличение лимита на число ошибок до 100.

Библиотечная карта

¤ *биб //тело*

переключает компилятор на трансляцию текста из файла *спр//тело*, где *спр* – основа внешних имен, переданная пятым параметром компилятору.

Все эти режимы традиционны для компиляторов в системе "Эльбрус". Рассмотрим специфические режимы клу-компилятора (приведены полное и краткое имя режима, его значение по умолчанию и семантика):

ген (*г*, **истина**) – управление генерацией кода. Компилятор состоит из двух просмотров: на первом выполняются все виды анализа и контроля и формируется внутреннее представление (*дерево*), по которому на втором просмотре генерируется код. Управляющая карта ¤ *уст*0 *ген* отключает генерацию кода. Если на первом просмотре обнаружены ошибки, генерация кода (при любом состоянии режима *ген*) не выполняется;

рус (**ложь**) – переключение на русскую лексику служебных слов. Как и в системе Паскаль-Эльбрус, в клу-системе введена русская лексика входного языка (см. приложение 4). Лексика служебных слов управляется режимом *рус*; для предопределенных идентификаторов (например, для имен ситуаций) русская лексика доступна одновременно с английской;

откр (*о*, **ложь**) – управление реализацией процедур и итераторов. В режиме ¤ *уст откр*, до его отключения, все компилируемые процедуры и итераторы (например, операции кластера) реализуются как *открытые*, т.е. подстановкой кода в место их вызова. Применение этого режима в системе Клу-Эльбрус существенно повышает эффективность программы. Управляющая карта ¤ *уст откр* может задаваться только в тексте описания реализации или полного описания модуля, так как открытая подстановка всегда связана с конкретной реализацией модуля (при другой реализации объем модуля может быть существенно иным, а его открытая подстановка – нецелесообразной);

стат (**ложь**) – управление режимом сборки программы. Если модуль используется в клу-программе, для которой предполагается статическая сборка, то он должен быть оттранслирован в специальном режиме ¤ *уст стат*. В этом режиме объектный код модуля не генерируется, т.е. выполняется только первый просмотр, а вместо объектного кода в библиотеку

записывается внутреннее представление модуля, которое затем используется при статической сборке. По умолчанию статическая сборка отключена, и выполняется динамическая сборка модулей при исполнении клу-программы.

Пример вызова клу-транслятора в диалоге.

Входной текст вводится с терминала. Набор пользователя выделен курсивом. Пустая посылка обозначена символом *.

```
тр э : //клу к ксн//клу//биб//код
<<< транслятор Клу-Эльбрус. версия 10.08.87. дата 17.09.87 >>>
00000000 ¤ уст  vt
00001000 p = proc (x : int)  returns (int)
00002000              return (x + 1)
00003000 end p
00004000 *
         трансляция модуля закончена
         размер входного текста (в строках) = 4
         размер кода (в словах) = 6
  <<< конец работы Клу-транслятора >>>
```

Пример неверного вызова клу-транслятора:

тр //клу : //клу к //код

Трансляция немедленно прекращается; на терминал выдается сообщение "были синтаксические ошибки!", а в файл виртуального вывода – следующая информация:

```
<<< транслятор Клу-Эльбрус, версия . . . дата . . . >>>
  неверно задан файл входного текста!
  трансляция прекращена
  количество ошибок = 1
  номер последней строки = 0
```

3.11.2. Библиотека модулей и разработка программ. Библиотека модулей в клу-системе предназначена для хранения информации о модулях, контроля связей между модулями и поддержки пошаговой разработки программ. В системе "Эльбрус" функции хранения и именования модулей обеспечиваются общим архивом (п. 1.23), который может состоять из сети справочников, определяющих внешние имена связанных с ними внешних объектов. В архиве может храниться, в частности, расширенный файл объектного кода, содержащий всю необходимую информацию об оттранслированном модуле или контексте компиляции на языке Клу-Эльбрус. Поэтому для системы Клу-Эльбрус разработки какой-либо специализированной базы данных не потребовалось, и в качестве библиотеки модулей Клу используется общий архив. Стиль и способы работы с библиотекой зависят от программиста. Он может, например, создавать все файлы и справочники, составляющие структуру библиотеки, средствами стандартного диалога МВК "Эльбрус" (п. 1.33) и передавать клу-транслятору внешние имена файлов текста и объектного кода в качестве параметров. При этом компилятор обеспечивает установку внутренних системных ссылок между

файлами (например, ссылки из файла объектного кода на файл его контекста компиляции). Приведем пример такого стиля работы с системой Клу-Эльбрус как с обычным компилятором на языке стандартного диалога.

```
> кн ксн//клу
> ссп биб
  биб
> кн ксн//клу//биб
  соз тстек
  тстек
> ред тстек
 редактор « эль » версия . . . дата . . .
: вс 00001 + 1000/
00001000 ¤ уст л
00002000 stack = cluster is create, push, pop
00003000 rep = array [int];
00004000 create = proc ( ) returns (cvt)
00005000           return ( rep $ new ( ) )
00006000           end create;
00007000 push = proc  (s : cvt, x : int)
00008000               rep $ addh (s, x)
00009000         end push;
00010000 pop =  proc (s : cvt) returns (int) signals (empty)
00011000         if rep $ size (s) = 0 then signal empty
00012000         else return (rep $ remh (s))
00013000         end end pop
00014000 end stack
00015000/
: к
нормальное завершение редакции
число блоков файла = 1
> сох
> тр //тстек к //кстек : ксн//клу//код
  создан файл кстек
> соз тпроц
  тпроц
> ред тпроц
  редактор « эль » версия . . . дата . . .
: вс 00001 + 1000/
00001000 ¤ уст л
00002000 stack = ксн // клу // биб // кстек
00003000 use = proc ( ) returns (int)
00004000 s : stack := stack $ create ( );
00005000 stack $ push (1);
00006000 return (stack $ pop ( ))
00007000 end use
00008000/
: к
```

нормальное завершение редакции
число блоков файла = 1
> *сох*
> *тр //троц к //кпроц : ксн//клу//код*
печ ацпу

В результате этого фрагмента диалога будет создана библиотека с внешним именем *ксн//клу//биб* и в ней – файлы текстов и кодов кластера *stack* (*//тстек* и *//кстек*) и использующей его процедуры *use* (*//троц* и *//кпроц*). Контекст компиляции, содержащий ссылку на код модуля *stack*, задается в текстовом файле *//троц* перед текстом процедуры *use*. На АЦПУ будут распечатаны протоколы работы клу-транслятора.

Поддержка пошаговой разработки программ. Как видно из приведенного примера, система Клу-Эльбрус не навязывает программисту какой-либо жесткой дисциплины разработки клу-программ (сверху вниз, снизу вверх и т.п.). Как дополнительная возможность, в системе имеются *директивы управления библиотекой*. Они делятся на четыре группы.

1. *Директивы работы с архивом* аналогичны директивам стандартного диалога и позволяют, не выходя из клу-системы, выполнить необходимые действия над архивом (библиотекой):

¤ *кн внешнее имя* – установка архивного контекста (внешнее имя задает справочник). Например: ¤ *кн ксн//клу*;

¤ *соз имя файла* – создание файла в текущем контексте; например, ¤ *соз код*1;

¤ *ссп имя справочника* – создание справочника в текущем контексте; например, ¤ *ссп биб*;

¤ *уда внешнее имя* – удаление файла или справочника с заданным внешним именем; например: ¤ *уда //код*1, ¤ *уда //биб*;

¤ *обм внешнее имя*1 *внешнее имя*2 – обмен (перестановка) внешних ссылок: первое имя устанавливается на объект, именовавшийся вторым именем, второе – на объект, именовавшийся первым именем; например: ¤ *обм //код*1 *//код*2.

2. *Директивы установки имен*

¤ *код внешнее имя* – установка внешнего имени файла объектного кода обрабатываемой единицы компиляции. Например: ¤ *код //код*1

3. *Директивы установки контекстов компиляции*

¤ *комп внешнее имя*1, . . . , *внешнее имя n* – установка контекста компиляции для транслируемого модуля. Внешние имена должны ссылаться на файлы объектного кода контекстов компиляции. Для компилируемого модуля становятся доступными (глобальными) идентификаторы, входящие в указанные контексты компиляции, и именуемые ими объекты (константы, типы, модули). Конфликты имен разрешаются в пользу контекста, внешнее имя которого задано в списке *правее*. Например: ¤ *комп //глобконт*, *//конт* . Такая форма директивы *комп* позволяет организовать любую контекстную связь модулей – иерархическую, сетевую и т.п. – и использовать в любом модуле любой другой модуль, имеющийся в библиотеке. Отметим, что *контекст компиляции* (единица именования объектов и организации программы) и *архивный контекст* (справочник, содержащий

имена файлов текста и кода единиц компиляции) непосредственно не связаны между собой. Как правило, в практике программирования используется одно-, двух- или трехуровневый архивный контекст (большее число уровней становится неудобным), а связь контекстов компиляции может быть значительно более сложной.

4. *Иформационные директивы*

¤ *л внешнее имя* – выдача информации об архивном объекте. Если это справочник, выдается список его элементов; контекст компиляции – список входящих в него имен и соответствующих им объектов; файл текста или кода модуля – краткая информация о нем. Если вместо внешнего имени указан ключ *кн*, то подразумевается текущий архивный контекст, *комп* – текущий контекст компиляции. Например: ¤ *л //код*1;

¤ *инф* [*имя директивы*] – школа (информация о директивах). Если указано имя директивы, выдается информация о ней. Если задан ключ *общ*, выдается общая информация о назначении библиотеки модулей и способах работы с ней. Если ни ключ, ни имя директивы не заданы, выдается список директив.

П р и м е р использования директив управления библиотекой Клу.

1. *Трансляция контекста компиляции*

```
> тр э : ксн//клу//код
««транслятор клу-эльбрус. версия . . . дата . . .»»
¤ уст vt
¤ кн ксн//клу//биб
* бибклу: установлен архивный контекст ксн//клу//биб
¤ соз линалг
* бибклу: в контексте создан файл линалг
  eps = 1 e – 15;
  vector = // вект;
  matrix = // матр
¤ код линалг
* бибклу: установлен файл кода //линалг
трансляция контекста закончена
размер входного текста (в строках) = 7
размер кода (в словах) = 2
* бибклу: объектный код записан в файл //линалг
« конец работы клу-транслятора »
```

2. *Трансляция независимой процедуры в заданном контексте компиляции*

```
> тр э  : ксн//клу//код
««транслятор клу-эльбрус. версия . . . дата . . .»»
¤ уст vt
¤ кн ксн//клу//биб
* бибклу: установлен архивный контекст ксн//клу//биб
¤ комп // станд // линалг
* бибклу: установлен контекст компиляции //станд
* бибклу: установлен контекст компиляции //линалг
¤ соз матрвект
* бибклу: в контексте создан файл матрвект
```

```
mult_ mv = proc (m : matrix, v : vector) returns (vector)
           signals (wrong_ size)
    res : vector := vector $ create ( );
    if matrix $ line_ size (m) = vector $ size (v)
    then signal wrong_ size
    else for i : int in int $ from_ to (1, matrix $ column_ size (m))
        do s : real := 0.0;
           li : vector := matrix $ line (i);
           for j : int in vector $ indexes (v) do
            s := s + li [ j ] * v[ j ]
           end;
           vector $ addh (res, s)
           end
    end
end mult_ mv
¤ код //матрвект
* библу: установлен файл кода//матрвект
трансляция модуля закончена
размер входного текста (в строках) = 19
размер кода (в словах) = 30
* библу: объектный код записан в файл //матрвект
⟪ конец работы клу-транслятора ⟫
```

В примере представлено два последовательных вызова клу-системы. В первом создается и транслируется контекст компиляции *//линалг* (линейная алгебра), состоящий из ссылок на кластеры *matrix* и *vector* (их описания здесь не приводятся). Во втором вызове системы компилируется независимая процедура *mult_ mv* (умножение матрицы на вектор). При этом устанавливается в качестве контекста компиляции объединение двух контекстов: *//станд* (стандартный) и *//линалг* (линейная алгебра). После выполнения каждой директивы управления библиотекой система Клу-Эльбрус выдает информационное сообщение, начинающееся именем *бибклу*.

Управление реализациями модулей. Каждый модуль на языке Клу имеет интерфейс и одну или несколько реализаций. Интерфейс и реализация могут храниться в одном или в разных файлах объектного кода. В контексте компиляции, связывающем другие модули с модулем m, может указываться как внешнее имя файла объектного кода интерфейса, так и имя файла кода реализации:

m = //инткод

или

m = //реалкод

Перед исполнением программы должно быть известно, какая реализация модуля m будет использоваться. Это определяет клу-система по внешнему имени, указанному в контексте компиляции. Если это имя кода реализации модуля (*//реалкод*), то берется указанная реализация модуля. Если же это имя кода интерфейса (*//инткод*), то в этом же файле кода должен находиться и код реализации. При трансляции полного описания

модуля

m = **proc** (*x* : **int**) . . . **end** *m*

клу-система автоматически размещает код интерфейса и реализации модуля в одном файле. При пошаговой разработке модуля (раздельной компиляции его интерфейса и реализации) следует указывать для интерфейса и реализации в директиве *тр* одно и то же имя файла объектного кода (при этом клу-система соединяет интерфейс и реализацию и получает в файле *//код* объектный код полного описания модуля):

тр //интерфейс : ксн//клу//код к //код
тр //реализация : ксн//клу//код к //код

Если требуется сменить имя реализации, уничтожить реализацию и т.п., то следует воспользоваться директивами *соз*, *обм*, *уда* системы Клу-Эльбрус или аналогичными директивами стандартного диалога МВК "Эльбрус" (однако последнее требует выхода из клу-системы).

3.11.3. Особенности сборки и исполнения программ. Описание каждого модуля на языке Клу отображается в файл объектного кода модульного объекта МВК "Эльбрус" (п. 1.31). После трансляции очередного модуля в библиотеке находятся файлы объектного кода модулей, связанных между собой отношениями вызова.

Сборка и запуск программы. Клу-система имеет два режима сборки исполняемой программы: статический и динамический.

Динамическая сборка выполняется по умолчанию. В режиме динамической сборки каждому модулю соответствует либо один, либо два файла объектного кода (интерфейса и реализации). Никаких предварительных действий перед исполнением программы не требуется. Все модули клу-программы в режиме динамической сборки исполняются в контексте *ядра* клу-системы – модуля, содержащего ссылки на вводные и выводные файлы, указатели глобальных областей памяти, метки стандартных процедур и другие системные объекты. В терминах модульных объектов языка Эль-76 (п. 1.31) ядро клу-системы является *формальным контекстом* каждого из ее независимых модулей. Поэтому запуск клу-программы в режиме динамической сборки осуществляется через специальную программу – "пускач", которая создает экземпляр ядра и запускает головную процедуру клу-программы в контексте ядра:

исп //пуск (*ввод*, *вывод*, $//p, x_1, \ldots, x_n$) ,

где *//пуск* – внешнее имя файла объектного кода программы-пускача, *ввод* и *вывод* – входной и выходной файлы программ, $//\ p$ – внешнее имя файла объектного кода головной процедуры; $x_1, \ldots, x_n$ – ее фактические аргументы. Например:

исп //пуск (э, э, *//нод*, 121, 55)

При исполнении программы будут созданы динамические экземпляры модулей, прямо или косвенно вызваемых из модуля *//нод*. Динамическую сборку рекомендуется использовать при автономной отладке отдельных модулей.

Статическая сборка, обеспечивающая получение более эффективного кода, должна быть выполнена до исполнения программы. Перед статичес-

кой сборкой все модули программы должны быть оттранслированы в специальном режиме ¤ *уст стат*. Для выполнения статической сборки предназначена директива клу-системы:

¤ *сборка //p*

где *//p* – внешнее имя файла объектного кода головной процедуры клу-программы. При выполнении этой директивы клу-система анализирует все связи модулей по вызову, осуществляет их необходимые конкретизации и генерирует единый файл объектного кода, состоящий из объектных кодов всех конкретных модулей. Полученная программа имеет следующий список фактических аргументов:

(*ввод*, *вывод*, $x_1, \ldots, x_n$)

где *ввод* и *вывод* – ссылки на входной и выходной файлы, а $x_1, \ldots, x_n$ – аргументы головной процедуры. Структура полученной программы может быть выражена на языке Эль-76 следующим образом:

программа = **проц** (*ввод*, *вывод*, $x_1, \ldots, x_n$)
  начало
    конст *ядро* # *//интклу* = (*//реалклу*, *ввод*, *вывод*);
    % подключение ядра клу-системы
    % – – – описание конкретного модуля 1
    . . .
    % – – – описание конкретного модуля
    % – – – тело программы:
    p ($x_1, \ldots, x_n$)
  конец

Запуск программы, полученной после статической сборки, осуществляется в следующей форме:

исп //прог (*ввод*, *вывод*, $x_1, \ldots, x_n$)

где *//прог* – внешнее имя файла объектного кода программы. Аргументы x_i – те же, что и при динамической сборке.

Динамическая диагностика и отладка. Динамическое поведение клу-программы ничем принципиально не отличается от программы на любом другом языке (например, на Эль-76 или на Паскале). При динамической ошибке выдается символьная распечатка стека (п. 2.5.4).

ГЛАВА 4

СПЕЦИАЛИЗИРОВАННЫЕ СИСТЕМЫ ПРОГРАММИРОВАНИЯ ДЛЯ МВК "ЭЛЬБРУС"

Эта глава посвящена описанию входных языков и возможностей специализированных систем программирования АБВ-Эльбрус, Снобол-Эльбрус, Рефал-Эльбрус, ДИАШАГ, Форт-Эльбрус и Модула-2-Эльбрус, созданных в Ленинградском университете. Каждая из них ориентирована на определенный класс задач: система АБВ – на разработку трансляторов; Снобол и Рефал – на задачи символьной обработки и искусственного интеллекта; ДИАШАГ – на пошаговую разработку, отладку программ и обучение программированию в диалоговом режиме; Форт – на определение и использование в режиме диалога простых расширяемых языков для задач системного программирования; Модула-2 – на создание модульных программных систем (главным образом для управления устройствами, ресурсами и процессами).

Эти системы, в силу своей ориентации и практических свойств, покрывают такие потребности программистов, удовлетворение которых средствами базовой системы Эль-76 и других производственных универсальных систем программирования (например, Паскаль-Эльбрус и Клу-Эльбрус) в полной мере невозможно или нецелесообразно.

Операции, характерные для символьной обработки (построение и анализ списочных структур, сравнение с образцом, замена по образцу и т.п.), не имеют прямой аппаратной поддержки в системе "Эльбрус" и, соответственно, не представлены в базовом языке Эль-76. Ограничения на работу с адресной информацией в системе "Эльбрус" не затрудняют реализацию операций над списочными структурами, так как имеется возможность размещения элементов списка в векторе и работы с их индексами как с относительными адресами. Более того, динамические свойства системы "Эльбрус" (см. гл. 6) позволяют реализовать языки этого класса эффективнее и полнее, чем на традиционных ЭВМ. Например, система Снобол-Эльбрус значительно превосходит по своим эксплуатационным характеристикам известные отечественные и зарубежные снобол-системы [71].

Диалоговые возможности системы программирования не должны исчерпываться лишь вводом транслируемого текста с терминала. В этом отношении система ДИАШАГ является значительно более гибкой, чем традиционные трансляторы, так как позволяет вводить текст частями (шагами),

корректировать его, расширять, автоматически перетранслировать и частично выполнять.

Язык и система АБВ представляют большой интерес прежде всего своим практическим программистским подходом к выражению семантики языков программирования, которые могут быть реализованы с помощью АБВ-системы. Исследования по системе АБВ-Эльбрус показали глубокое сходство принципов и базовых понятий языка АБВ и системы "Эльбрус"; в результате была создана эффективная компиляционная реализация языка АБВ.

Стимулом к разработке системы Модула-2-Эльбрус явился большой интерес к языку Модула-2 в последние годы и, кроме того, наличие готовой базы для разработки – системы Паскаль-Эльбрус (как известно, язык Модула-2 создан на основе языка Паскаль).

Эти, лишь кратко затронутые, особенности указанных систем и их возможности, рассматриваемые в настоящей главе, позволяют надеяться на практический интерес программистов к этим системам, который может быть удовлетворен при более массовом распространении системы "Эльбрус".

4.1. Расширяемый язык АБВ и система АБВ-Эльбрус

4.1.1. Обзор языка АБВ и его реализаций. АБВ [45] – расширяемый язык программирования, созданный С.С. Лавровым, Е.Н. Капустиной и М.И. Селюном. По своему назначению АБВ – базовый язык для разработки трансляторов. Его назначение и возможности рассматриваются в работах [37, 41, 43, 44]. *Действия* (конструкции) языка АБВ делятся на три группы – *анализатор* (аппарат работы со строками), *базу* (средства именования и управления) и *вычислитель* (конструкции низкого уровня для выполнения вычислений, настраиваемые на конкретную ЭВМ).

В системе программирования, создаваемой с помощью базового языка АБВ, сам язык АБВ, по замыслу авторов, должен использоваться как инструментальный и как выходной язык. Анализатор служит для лексического и синтаксического анализа текстов на входном языке и для генерации выходного текста на языке АБВ. База применяется для выражения семантики аппарата имен, средств типизации и управления входного языка и содержит ряд общих понятий и конструкций, удобных для этой цели. Вычислитель используется как подмножество выходного языка для трансляции действий над числовыми и логическими значениями, а также для выполнения вычислений в самих АБВ-программах.

Из соображений упрощения трансляции для конструкций языка АБВ принят скобочный синтаксис (см. приложение 5). Однако его расширяемость дает возможность ввести любое удобное для записи программ синтаксическое расширение (например, алголоподобный синтаксис), реализовав его компилятор в конструкции *ядра* языка АБВ средствами ядра.

Рассмотрим основные особенности языка АБВ.

Б а з а. Основные понятия базы – *объект*, *имя*, *действие*. Объект создается операцией базы, существует глобально (пока на него имеются ссылки) и состоит из *вида* (типа) и *значения*. Вид хранится вместе со значением. В ядре имеются следующие собственные (стандартные) виды:

ДЕЙСТВ – *действие* (исполняемая единица программы); УКАЗ – *указатель* на объекты произвольного типа; СТРОКА – *строка* символов; ФОРМАТ – *формат* для вывода или преобразования значений к символьной форме; ЦЕЛ – *целое* число произвольной длины; ЦЕЛ1 – *целое единичной длины* (длиной в одно слово); ПРИБ – *приближенное* число произвольной длины; ЛОГ – *логическое*; ШКАЛА – *логический вектор* произвольной длины; ПОЛЕ – область оперативной или внешней *памяти*. Виды ДЕЙСТВ и УКАЗ относятся к базе, СТРОКА и ФОРМАТ – к анализатору, остальные – к вычислителю. Например, операция (ОБЦ1 щц 1) создает новый объект вида ЦЕЛ1 со значением 1. Все собственные виды считаются значениями вида ДЕЙСТВ.

Имя – это совокупность способов обращения к объекту. Имя может состоять из нескольких *синонимов*, наиболее распространенным из которых является *идентификатор*. Оператор *ввести синоним* (ВСИН Т и А) – аналог описания переменной – создает объект вида Т с неопределенным значением и связывает с ним синоним – идентификатор А. Область существования этого синонима ограничена ближайшим объемлющим *последовательным действием*:

(ПОСЛ ...(ВСИН Т и А) ...)

– аналогом блока. С идентификатором А в каждом последовательном действии связан стек именуемых объектов. Оператор *ввести синоним* помещает в вершину стека новый объект. Обратное действие выполняет оператор *ликвидировать синоним* (ЛИКВ и А). Поскольку операторы ВСИН и ЛИКВ являются исполняемыми действиями, которые могут быть частью условных и циклических конструкций, смысл идентификаторов в общем случае определяется динамически. В остальном правила образования контекста идентификаторов аналогичны блочной структуре.

Операция *сравнение имен* (РАВИМ А В) выдает результат "истина", если А и В представляют одно и то же имя (т.е. ссылаются на один и тот же объект).

Оператор *присвоить* (ПРИСВ А В) присваивает значение объекта А объекту В. Например:

(ВСИН ЦЕЛ1 *и* i) (ПРИСВ (ОБЦ1 щц 0) i)

Операция (ОБУК А) создает объект-*указатель*, значением которого является ссылка на объект А. Последовательность операторов

(ВСИН УКАЗ и Р) (ПРИСВ (ОБУК А) Р)

присваивает это ссылочное значение объекту-указателю Р. Обратная операция – *косвенное обозначение* (ЗНАЧУК Р) возвращает ссылку на объект А.

Результат выполнения последовательного действия – *составной объект* – структура из объектов, именуемых его локальными идентификаторами. Вид составного объекта – создавшее его последовательное действие. Рассмотрим схему генерации составного объекта на следующем примере:

(ВСИН ДЕЙСТВ и КОМПЛ)
(ПРИСВ д (ПОСЛ (ВОЗОБН) (ВСИН ПРИБ и *re*)
(ВСИН ПРИБ и *im*)) КОМПЛ)

Конструкция д... – *изображение действия* – задает значение объекта КОМПЛ вида ДЕЙСТВ. Его можно рассматривать как обозначение типа "комплексное число". Роль генератора составных объектов вида КОМПЛ играет *операция прерывание* (ПРЕР КОМПЛ). Последовательность операторов

(ВСИН КОМПЛ и z) (ПРИСВ (ПРЕР КОМПЛ) z)

создает новый объект вида КОМПЛ, именуемый идентификатором z. Оператор *возобновить* (ВОЗОБН) указывает, что после завершения действия КОМПЛ следует возобновить вызвавшее действие, т.е. обозначает процедурную связь с действием КОМПЛ. Компоненты составного объекта (атрибуты) именуются с помощью *составных обозначений*, например (АТР и *re* z). Комплексное значение 1.0 + 2.0i присваивается объекту z следующим образом:

(ПРИСВ (ОБПР п 1 п 1.0) (АТР и *re* z))

(ПРИСВ (ОБПР п 1 п 2.0) (АТР и *im* z))

Запуск атрибутов-действий. Важным частным случаем атрибута составного объекта является атрибут вида ДЕЙСТВ (действие). Если атрибут-действие запускается операцией *прерывание*, то при этом устанавливается контекст идентификаторов, присущий всем последовательным действиям, исполнение которых привело к созданию составного объекта. В терминах языка Эль-76 (п. 1.31) последовательное действие есть модульный объект из одного интерфейса (в языке АБВ нет скрытых атрибутов); объемлющие последовательные действия задают формальный контекст этого модульного объекта. Процедурные атрибуты составного объекта в языке АБВ были попыткой ввести в этот язык механизм, подобный тому, который впоследствии получил название абстрактных типов данных.

П р и м е р. Моделирование на языке АБВ параметризованного АТД "стек".

```
(ПОСЛ
      (ВСИН ДЕЙСТВ и Т) % параметр-тип
      (ВСИН ДЕЙСТВ и СТЕК)
      (ПРИСВ д
        (ПОСЛ (ВОЗОБН)
           (ВСИН ДЕЙСТВ и ЭЛЕМ) % скрытая процедура
           (ПРИСВ д (ПОСЛ (ВОЗОБН) (ВСИН Т и ЭЛ)
                          (ВСИН УКАЗ и СЛЕД)) ЭЛЕМ)
           (ВСИН УКАЗ и nil) (ПРИСВ (ОБУК nil) nil)
           (ВСИН УКАЗ и ВЕРШИНА) (ПРИСВ nil ВЕРШИНА)
           (ВСИН ДЕЙСТВ и ПОЛОЖИТЬ) % – – операция "положить"
           (ВСИН Т и Х) % – – аргумент операции "положить"
           (ПРИСВ д % – – реализация операции "положить"
              (ПОСЛ (ВОЗОБН)
                    (ВСИН УКАЗ и ТЕК)
                    (ПРИСВ (ОБУК (ПРЕР ЭЛЕМ)) ТЕК)
                    (ПРИСВ Х (АТР и ЭЛ (ЗНАЧУК ТЕК)))
                    (ПРИСВ ВЕРШИНА (АТР и СЛЕД (ЗНАЧУК ТЕК)))
                    (ПРИСВ ТЕК ВЕРШИНА))
              ПОЛОЖИТЬ) % – – конец реализации "положить"
```

```
            (ВСИН ДЕЙСТВ и ВЗЯТЬ) % – – операция "взять"
            (ВСИН ДЕЙСТВ и ИСЧЕРПАНИЕ) % ситуация в операции
                                        % взять
            (ПРИСВ д % – – реализация операции "взять"
                (ПОСЛ (ВОЗОБН)
                    (АЛЬТ (РАВИМ (ЗНАЧУК ВЕРШИНА) nil)
                          (ПРЕР ИСЧЕРПАНИЕ)
                          (ЗАВЕРШ (АТР и ЭЛ (ЗНАЧУК ВЕРШИНА)))))
                ВЗЯТЬ) % – – конец реализации "взять"
)СТЕК) % – – – конец определения АТД "стек"
% – – использование стека:
% конкретизация стека типом элемента ЦЕЛ1
(ПРИСВ ЦЕЛ1 Т)
% создание пустого стека объектов типа ЦЕЛ1
(ВСИН СТЕК и СТЕКЦЕЛ) (ПРИСВ (ПРЕР СТЕК) СТЕКЦЕЛ)
% использование операций над стеком
(ПРИСВ(ОБЦ1 щц 1) (АТР и Х СТЕКЦЕЛ))
(ПРЕР (АТР и ПОЛОЖИТЬ СТЕКЦЕЛ))
(ВСИН ЦЕЛ1 и i) (ПРИСВ (ПРЕР (АТР и ВЗЯТЬ СТЕКЦЕЛ)) i)
% вывод результата
(ВЫВОД (ПЦ1С (ЗНЦ1 i) (ЗНФ ФЦ1)) (ЗНФ ФТ))
)
```

Определяется последовательное действие СТЕК, которое является и описанием АТД "стек", и генератором объектов этого типа. Конкретное представление стека – связанный список из элементов типа ЭЛЕМ и указатель ВЕРШИНА. Операция ПОЛОЖИТЬ имеет аргумент Х, который описывается как атрибут объекта СТЕК вместе с самой операцией (в языке АБВ в операции *прерывание* действию-реакции не могут быть переданы аргументы). Операция ВЗЯТЬ выдает объект из вершины стека (результат задается оператором *завершить*). Операция ВЗЯТЬ можеть закончиться ситуацией ИСЧЕРПАНИЕ, реакция на которую должна быть задана оператором присваивания

(ПРИСВ РЕАКЦИЯ (АТР и ИСЧЕРПАНИЕ СТЕКЦЕЛ))

в использующем действии. В теле операции *взять* использовано *альтернативное действие* (АЛЬТ) – аналог условного оператора. Операции ПОЛОЖИТЬ и ВЗЯТЬ запускаются как атрибуты-действия объекта СТЕКЦЕЛ.

Управление. Основные управляющие конструкции языка АБВ – *последовательное* (ПОСЛ), *параллельное* (ПАР) и *альтернативное* (АЛЬТ) *действия* и *операция прерывание* (ПРЕР). Операция *прерывание* (ПРЕР Х) позволяет организовывать любой вид связи по управлению – вызов процедуры, переход по метке или по ситуации, вызов сопрограммы. Первая из них возникает, если в реакции Х выполнен оператор *возобновить* (пример приведен выше). Если же оператор *возобновить* не был выполнен в реакции Х, происходит *прекращение* прерванного действия: управление передается оператору, следующему за тем, в котором реакция *X* была определена. Например, цикл

repeat S until B

моделируется последовательностью операторов языка АБВ:

(ВСИН ДЕЙСТВ и *repeat*) (ПРИСВ д ( ) *repeat*)
S (АЛЬТ В ( ) (ПРЕР *repeat*))

Сопрограммная связь получается, если два составных объекта С1 и С2 взаимодействуют с помощью операций ПРЕР (АТР и УПР С2)) и (ПРЕР АТР и УПР С1)), где УПР – атрибут вида ДЕЙСТВ, значение которого указывает текущую точку передачи управления.

Средства расширения. Операция преобразование строки в действие (ПСД *S*) осуществляет преобразование текста последовательного действия, заданного в виде строки *S*, во внутреннюю форму для его последующего запуска, т.е. динамическую трансляцию и расширение программы. Строка *S* может быть получена любым способом, например вводом. Новое действие, расширяющее АБВ-программу, локально в объемлющем последовательном действии:

(ПОСЛ . . . (ВСИН ДЕЙСТВ и РАСШИРЕНИЕ)
(ПРИСВ (ПСД (ВВОД)) РАСШИРЕНИЕ) . . . (ПРЕР РАСШИРЕНИЕ) . . .)

Анализатор. Основное понятие анализатора – строка. *Строка* – это последовательность *элементов* (символов или подстрок), взятых в парные кавычки: 'С' ТР' О' КА" – строка из четырех элементов С, 'ТР', О и 'КА' и 10 символов (считая внутренние кавычки). Анализатор, в отличие от базы, работает только со значениями-строками. Операция (ЗНСТР С) осуществляет переход от объекта-строки к его значению. Операция ЧЭЛ и ЧСИМ определяют число элементов и символов строки. *Операторы анализа* строки – *выделить* (ВЫД) и *найти вхождение* (ВХОЖД) – разбивают строку на части и присваивают ссылки на них стандартным объектам ГОЛОВА и ХВОСТ, а признак успешного выполнения оператора – стандартному объекту УСПЕХ вида ЛОГ. Оператор (ВЫД щц 3 'С' ТР' О' КА") выделяет три первых элемента строки: значением объекта ГОЛОВА становится выделенная подстрока 'С'ТР' О', объекта ХВОСТ – остаток строки "КА". Оператор (ВХОЖД 'О' 'С' 'ТР' О 'КА") находит первое вхождение первой строки во вторую и присваивает объекту ГОЛОВА часть строки до вхождения 'С' ТР", а объекту ХВОСТ – часть строки после вхождения 'О'КА". *Сравнение строк* выполняется операцией базы (РАВС С1 С2), где С1 и С2 – объекты-строки.

Операция *взять в кавычки* (КАВЫЧ) и *снять кавычки* (СНКАВ) предназначены для создания и анализа строк. *Операция сцепление* (СЦЕПЛ) – конкатенация строк. Например, оператор:

(ПРИСВ (ОБСТР (СЦЕПЛ (КАВЫЧ '*begin*')
(СЦЕПЛ 'x: = 1' (КАВЫЧ '*end*')))) ТЕКСТ)

присваивает объекту ТЕКСТ значение-строку "*begin*'*x*: = 1'*end*"; последовательность операторов

(ВЫД щц 1 (ЗНСТР ТЕКСТ)) (ПРИСВ (ОБСТР (СНКАВ ГОЛОВА))
СИМВОЛ) присваивает объекту СИМВОЛ значение-строку '*begin*'.

Анализатор осуществляет также *ввод-вывод* строк. Операция (ВВОД) вводит символы одной строки (заданной без внешних кавычек) из стан-

дартного файла ввода и выдает значение-строку в качестве результата. Для ввода чисел и других значений предусмотрены операции преобразования строки: оператор (ПРИСВ (ПСЦ1 (ВВОД)) *i*) инициализирует объект *i* вида ЦЕЛ1 вводом. Обратные операции преобразования в строку используют форматы (значения вида ФОРМАТ): операция (ПЦ1С (ЗНЦ1 *i*) ф 35С ф) преобразует значение *i* в строку, состоящую из знака числа и пяти цифр. Для целых единичной длины имеется стандартный формат ФЦ1. Операция (ВЫВОД Х (ЗНФ ФТ)) выводит указанную строку по стандартному формату ФТ.

Пример. Проверка баланса служебных слов '*begin*' и '*end*' во входном тексте программы на алголоподобном языке. Служебные слова выделяются кавычками.

```
(ПОСЛ   (ВСИН СТРОКА н ТЕКСТ) (ПРИСВ (ОБСТР (ВВОД)) ТЕКСТ)
        (ВСИН СТРОКА н ЭЛТ) (ВСИН ЦЕЛ1 н ДЛИНА)
        (ПРИСВ (ОБЦ1 (ЧЭЛ (ЗНСТР ТЕКСТ))) ДЛИНА)
        (ВСИН ЦЕЛ1 н БАЛАНС) (ПРИСВ (ОБЦ1 щц 0) БАЛАНС)
        (ВСИН ДЕЙСТВ н ЦИКЛ) (ПРИСВ д (   ) ЦИКЛ)
        % – – начало цикла
        (ВЫД щц 1 (ЗНСТР ТЕКСТ))
        (ПРИСВ ГОЛОВА ЭЛТ) (ПРИСВ ХВОСТ ТЕКСТ)
        (АЛЬТ  (РАВС ЭЛТ (ОБСТР "begin"))
               (ПРИСВ (ОБЦ1 (Ц1С (ЗНЦ1 БАЛАНС) щц 1))БАЛАНС)
               ( АЛЬТ  (РАВС ЭЛТ (ОБСТР " end"))
                       (ПРИСВ (ОБЦ1 (Ц1В(ЗНЦ1 БАЛАНС) щц 1))
                       БАЛАНС)
        ( ПРИСВ (ОБЦ1 (Ц1В (ЗНЦ1 ДЛИНА) щц 1)) ДЛИНА)
        (АЛЬТ(ОБЛОГ(РАВЦ1 (ЗНЦ1 ДЛИНА) щц 0))( ) (ПРЕР ЦИКЛ))
        % – – конец цикла
        (АЛЬТ  (ОБЛОГ (РАВЦ1 (ЗНЦ1 БАЛАНС) щц 0))
               (ВЫВОД 'НЕТ БАЛАНСА' (ЗНФ ФТ))
               (ВЫВОД 'ВСЕ ПРАВИЛЬНО' (ЗНФ ФТ))))
```

Вычислитель состоит из операций для работы с числовыми и логическими значениями. Целые, приближенные числа и логические шкалы могут быть произвольной длины (точности), например ц 2 ц 123456789 – изображение целого длиной 2 слова. Любое значение может быть записано в *поле* – области оперативной или внешней памяти. Класс памяти характеризуется *уровнем* поля: 0 – оперативная, > 0 – внешняя. Поле создается и освобождается действиями базы. Для группировки действия вычислителя и управления ими введены *составное действия вычислителя* (СОСВ), локальные *метки* СОСВ, *условные переходы* (УП) и *безусловные переходы* (БП).

Действия вычислителя могут быть оттранслированы в эффективный объектный код любой ЭВМ. Зависимость от конкретной ЭВМ выражена в виде совокупности параметров – именованных предопределенных величин: МАКЦ1Ч, МАКЦЧ, МАКПРЧ – максимальные значения типов ЦЕЛ1, ЦЕЛ и ПРИБ; МАКДЦ, МАКДПР и МАКДШ – максимальная длина целого, приближенного и шкалы; ЧРАЗЯ, ЧСИМЯ – число разрядов и символов в ячейке (слове); ЧСИМСТ, ЧСТСТ – число символов в строке и строк в странице при выводе; ЧУР – число уровней памяти.

П р и м е р. Вычисление факториала числа типа ЦЕЛ1.

```
(ПОСЛ  (ВСИН ПОЛЕ и f) (ПРИСВ (ПАМ щц 2 щц 0) f))
       (СОСВ (ЗАПЦ1 (ЗНПОЛЕ f) щц 0 (ЗНЦ1 (ПСЦ1 (ВВОД))))
             (ЗАПЦ1 (ЗНПОЛЕ f) щц 1 щц 1)
       МК    (ЗАПЦ1 (ЗНПОЛЕ f) щц 1
             (Ц1У (ЧИТЦ1 (ЗНПОЛЕ f) щц 0)
             (ЧИТЦ1 (ЗНПОЛЕ f) щц 1)))
       (ЗАПЦ1 (ЗНПОЛЕ f) щц 0
             (Ц1В (ЧИТЦ1 (ЗНПОЛЕ f) щц 0) щц 1))
       (УП (БОЛЦ1 (ЧИТЦ1 (ЗНПОЛЕ f) щц 0) щц 0) МК))
       (ВЫВОД (ПЦ1С (ЧИТЦ1 (ЗНПОЛЕ f) щц 1)(ЗНФ ФЦ1)) (ЗНФ ФТ)))
```

Этот простой пример демонстрирует "ассемблерный" стиль программирования на языке АБВ. Вместо присваиваний и переменных используется запись в поле (ЗАПЦ1) и чтение из поля (ЧИТЦ1). Слово 0 поля f содержит текущее значение числа, слово 1 – результат. В качестве управляющих конструкций используются локальная метка МК и условный переход УП. Смысл остальных операций: ЗНПОЛЕ – значение-поле; Ц1В, Ц1У, БОЛЦ1 – вычитание, умножение и операция "больше" для целых единичной длины.

Р е а л и з а ц и и я з ы к а АБВ. Первая (авторская) реализация – компилятор АБВ-БЭСМ-6 [70]. В нем, в строгом соответствии с описанием семантики языка АБВ, действия базы транслируются в интерпретационные вставки без каких-либо оптимизаций. В 1982 г. была создана первая версия системы АБВ-Эльбрус [41] со смешанной схемой трансляции: интерпретация аппарата имен и управления и компиляция действий вычислителя и анализатора. Ее составной частью является эффективная реализация аппарата имен, принципы которой не зависят от целевой ЭВМ [41]. Дальнейшие исследования в этом направлении показали, что для языка АБВ, несмотря на сложность его динамических механизмов, в системе "Эльбрус" может быть создан компилятор с хорошим качеством объектного кода, основанный на использовании аппаратных интерпретационных действий в системе "Эльбрус" (см. гл. 6).

4.1.2. Система программирования АБВ-Эльбрус [62] – компиляционная реализация языка АБВ. Система состоит из компилятора и подсистемы динамической поддержки. Форма обращения к компилятору, его управляющих карт и протокола работы унифицирована с системами Паскаль-Эльбрус (гл. 2) и Клу-Эльбрус (гл. 3).

Входной язык расширен собственным видом ПРИБ1 (*приближенное единичной длины*), изображениями его значений (пп 3.14) и необходимыми операциями (например, П1С – по аналогии с видом ЦЕЛ1; полностью это расширение описано в приложении 5). Для изображения строк вместо кавычек используются комбинации символов ('и') (во входном алфавите системы "Эльбрус" парные кавычки отсутствуют). Все конструкции языка реализованы без каких-либо ограничений, включая запуск процедурных атрибутов и параллельные действия. Параметры в данной реализации АБВ имеют следующие значения: МАКЦ1Ч = 9223372036854775807; МАКЦЧ – отсутствует; МАКПРЧ – отсутствует; МАКДЦ = 2 ** 20–1; МАКДПР =

= 2 ** 20–1; МАКДШ = 2 ** 16–1; ЧРАЗЯ = 63; ЧСИМЯ = 8; ЧУР = 2. Дополнительный параметр МАКПР1Ч = 7.237005577332262e75. Реализована арифметика многократной точности над целыми и приближенными числами. Поля могут иметь уровни 0 (оперативная память) и 1 (файлы на барабане). При запуске АБВ-программы на исполнение стандартные файлы ввода и вывода передаются ей в качестве параметров.

П р и м е р работы с системой АБВ-Эльбрус в диалоге.

```
    >кн ксн//абв
    >тр э://абвкод к //код
«« система абв-эльбрус. версия . . . дата . . . »»
00000000 ¤ уст  v т
00001000 (ПОСЛ  (ВСИН ПРИБ1_н s)  (ПРИСВ (ОБПР1 пп 0.0) s)
00002000        (ВСИН ПРИБ1_н x)  (ПРИСВ (ОБЛОГ_л 1) УСПЕХ)
00003000        (ПРИСВ_д (ПОСЛ (ВОЗОБН)  (ПРИСВ (ОБЛОГ_л 0)
                УСПЕХ))
00004000                 ВВОДПРЕР)
00005000        (ВСИН ДЕЙСТВ_н ЦИКЛ)  (ПРИСВ_д ( ) ЦИКЛ)
00006000        (ПРИСВ (ПСП1 (ВВОД)) x)
00007000        (АЛЬТ  УСПЕХ  % очередное число введено
00008000               (ПОСЛ  (ПРИСВ (ОБПР1 (П1С (ЗНПР1 s)
                                              (ЗНПР1 x))) s)
00009000                      (ПРЕР ЦИКЛ))
00010000               (ВЫВОД (СЦЕПЛ ('РЕЗУЛЬТАТ =')
00011000               (ППР1С (ЗНПР1 s) (ЗНФ ФПР1)))
00012000               (ЗНФ ФТ))))
00013000
      трансляция программы закончена
      размер входного текста (в строках) = 12
      размер кода (в словах) = 20
 «««  конец работы абв-системы »»
  >  исп//код  (э, э)
  >  1.0
  >  2.0
  >  3.0
  @
      результат = + 6.0000000000000000e + 00
```

Программа суммирует числа типа ПРИБ1 из входного файла до его исчерпания. Конец файла при выполнении операции ВВОД приводит к стандартному прерыванию ВВОДПРЕР, реакция на которое переопределяется как присваивание значения "ложь" стандартной логической переменной УСПЕХ. Полный список стандартных прерываний приведен в приложении 5. Конец вводимой с терминала строки – конец сообщения. Конец файла при вводе с терминала – набор символа "@" вместо стандартного приглашения ">".

4.2. Система Снобол-Эльбрус

Работа по реализации языка символьной обработки Снобол-4 [23] для МВК "Эльбрус" явилась логическим продолжением исследований по реализации языка АБВ [41]. Система Снобол-Эльбрус – первая отечественная реализация языка Снобол-4 – была создана в 1985 г. Н.Н. Болдиновой и А.Н. Смертиным. А.Н. Смертину принадлежит также концепция и реализация диалоговой подсистемы отладки программ на языке Снобол-4.

Особенности входного языка. Язык Снобол-4 – широко известный язык символьной обработки, реализованный на машинах IBM/360 и PDP-11. Эти реализации доступны в СССР. Автором языка Р. Грисуолдом разработана переносимая макрореализация Снобола-4.

Основные типы данных языка Снобол-4 – *строка* символов (STRING) и *образец* (PATTERN). Набор операций над строками включает: конкатенацию, сравнение с образцом, неявное присваивание сопоставленных фрагментов строк, замещение по образцу. Образцы могут иметь сложную древовидную и рекурсивную структуру; с их помощью можно описать любой контекстно-свободный синтаксис анализируемых строк. Предусмотрен механизм возвратов (FAILURE) при неудачном анализе. Из других типов данных языка Снобол-4 отметим ассоциативные таблицы (TABLE), независимо транслированные программы (CODE) и возможность определения структурных типов примитивной функцией ДАТА.

Язык Снобол-4 отличается крайней динамичностью. Тип переменной не фиксируется. Во время исполнения могут использоваться новые переменные с динамически вычисленными именами. Смысл и число аргументов любой операции (например, +) могут быть переопределены исполняемым оператором. Все эти свойства наряду со сложными операциями символьной обработки предопределяют интерпретационный характер реализации. Весьма важным для отладки снобол-программ свойством языка является наличие встроенных средств трассировки.

Система Снобол-Эльбрус. Эта система представляет собой интерпретатор полного языка Снобол-4 в соответствии с описанием [23]. Исходный текст снобол-программы преобразуется во внутреннюю форму (префиксную запись) и затем интерпретируется. Система Снобол-Эльбрус запускается следующим образом:

исп//снобол (*текст*, $f_1, \ldots, f_n$) ,

где // *снобол* – внешнее имя файла объектного кода снобол-системы; *текст* – файл исходного текста; $f_1, \ldots, f_n$ – файлы для ввода-вывода. При запуске системы в форме

исп//снобол (э)

входной текст снобол-программы вводится с терминала, и ввод-вывод и выдача протокола осуществляются также через терминальный файл.

Управление интерпретацией и отладка снобол-программ в диалоге осуществляются через *подсистему диалога*. Вход в нее происходит автоматически после успешной трансляции текста снобол-программы во внутреннее представление. Признак входа в подсистему диалога – сообщение:

***** диалог!

и приглашение =. Дальнейшее управление интерпретацией передается программисту. Директива *=И* (интерпретация; директивы для наглядности приводятся вместе с приглашением) запускает программу начиная с прерванного места. Директива *=К* – конец диалога. Программист имеет возможность запускать программу с заданного места, перед запуском программы указывать в ней контрольные точки (начало инструкции, вызов функции, возврат из функции, чтение и изменение значения простой переменной). Директива

=НА :М; 2

продолжает интерпретацию с инструкции, номер которой на 2 больше номера инструкции с меткой *М*. Директива

=О: *М*; 2; *EQ* (*A*, 1)

устанавливает контрольную точку в начале этой же инструкции. После выполнения директивы *=И* интерпретация будет приостановлена перед началом этой инструкции в том случае, если вычисление указанного выра жения будет успешным (т.е. значение переменной *А* будет равно 1). Останов по контрольной точке вызывает вход в диалог. Отмена контрольной точки выполняется директивой

=ОО: *М*; 2

Директива

= З: *А*; *NE*(*A*, 0)

устанавливает в качестве контрольной точки событие присваивания переменной *А*. Останов будет выполнен только в случае, если значение *А* окажется отличным от нуля. Директива

=В: *А* + *С*

вычисляет и выводит на терминал значение заданного выражения. Режимы работы снобол-системы устанавливаются директивой

=УСТ: *ТРАСС*

В данном случае установлен режим трассировки вызова основных системных процедур. Эта директива используется, главным образом, при отладке и проверке самой снобол-системы. Для этой же цели служит и ряд других отладочных директив: вывод содержимого системных таблиц, их атрибутов и т.д.

Пример работы системы Снобол-Эльбрус (пример принадлежит А.Н. Смертину):

>исп//снобол (//лекс, э)

```
Снобол-Эльбрус. версия 11.11.85 дата . . . время . . .
* лексикографическое упорядочение списка слов
1 output = 'расположить в алфавитном порядке слова списка:'
2 список = input
  максимальная длина слова в списке:
3 длина  = input
  * образцы: 1 – слово с запятой на конце; 2 – буква цепочки
```

```
 4 дайслово = (break (',') ','). слово
 5 дайбукву = len  (1). буква
 6 l1 алф = ', efhilnst'
 7 output = список
 8 длина = длина – 1;  lt (длина, 0)                  : s (финиш)
10 l3 список дайслово =                               : f (l5)
11 слово len (длина) дайбукву                         : f (l4)
12 $ буква = $ буква слово                            : (l3)
13 l4 коротслова = коротслова слово                   : (l3)
14 l5 список = коротслова; коротслова
16 счетчик  = 1
17 цикл алф дайбукву =     ;
18 список = список $ буква;  $ буква =      ;
20 счетчик = счетчик + 1; le (счетчик, 9)             : s (цикл)  f (l1)
22 финиш output = 'список слов в алфавитном порядке:'список
23 end
***** в исходной программе ошибок не обнаружено *****
      количество оттранслированных инструкций = 23
      номер первой исполняемой инструкции  = 1
      количество просмотренных строк  = 22
      время трансляции  = 00.55 с
***** диалог!
=u
расположить в алфавитном порядке слова списка:
   then,  if,    elif,  else,  fi,
   if,    fi,    else,  elif,  then,
   if,    fi,    then,  elif,  else,
   if,    then,  fi,    elif,  else,
   elif,  else,  fi,    if,    then,
список слов в алфавитном порядке: elif, else, fi, if, then,
=к
  $ $ $ $ $  интерпретация завершена
             время работы интерпретатора = 00.05 с
```

В программе использованы следующие конструкции языка Снобол-4:

– *конкатенация*: *A B* ;

– *образцы*: *break* (' , ') – часть строки до символа ' , ';

len (1) – часть строки длиной 1 символ;

– *неявное присваивание: len* (1). *буква* – сопоставленная строка (один символ) присваивается переменной *буква*;

– *динамическая переменная:* $ буква – переменная, имя которой является значением переменной *буква;*

– *сравнение с образцом, замещение, переход.* В инструкции 10 первое слово списка присваивается переменной *слово,* а в списке уничтожается. При неудачном исходе (*f*), т.е. если список пуст, выполняется переход на метку *l* 5;

– *ввод-вывод: x = input* – вводимая строка присваивается переменной *x*; *output = y* – строка *y* выводится.

Особенности реализации. В системе Снобол-Эльбрус применен ряд оригинальных методов реализации, основанных на использовании динамических возможностей системы "Эльбрус" [12]. Для представления строк используется модификация универсального ссылочного представления строк, примененного в системе АБВ-Эльбрус (п. 6.2.2). Для оптимизации сравнения с образцом применяется система внутренних эвристик. В системе используется новый эффективный алгоритм перевода из инфиксной формы в префиксную. Для представления данных применяются векторы переменной длины (пулы): объекты каждого типа размещаются в отдельном векторе, что упрощает выделение памяти и сборку мусора. Сборка мусора реализована как процесс, параллельный основной программе, и, как правило, это программа локализована в одном векторе. Для хранения промежуточных результатов используются два стека, которые позволяют избежать "замусоривания" памяти временными объектами. Благодаря всем этим свойствам реализации авторам удалось достичь высоких эксплуатационных характеристик системы. По оценкам, проведенным А.Н. Смертиным [71], система Снобол-Эльбрус требует в несколько раз меньше памяти и времени для выполнения одних и тех же программ и конструкций, чем другие известные реализации языка Снобол (оценка проведена с учетом разницы в быстродействии ЭВМ).

4.3. Система Рефал-Эльбрус

Язык программирования Рефал разработан, как и Снобол-4, для решения задач нечисленного анализа: обработки текстов, аналитических выкладок, автоматического доказательства теорем и т.п. Семантика и конструкции языка Рефал весьма нетрадиционны (в сравнении с процедурными языками программирования) и более близки к продукционным языкам и системам [1]. Программа на языке Рефал, хранящаяся в поле памяти рефал-системы, определяет, по существу, систему подстановки термов, под управлением которой выполняется вывод из исходного текстового выражения (помещаемого в поле зрения рефал-системы) другого текстового выражения. Несмотря на свою необычность, язык Рефал активно используется в СССР и реализован на многих ранее и ныне эксплуатируемых ЭВМ: М-220, Минск-32, БЭСМ-6, ЕС ЭВМ, СМ ЭВМ [6]. Поэтому разработка системы программирования Рефал-Эльбрус, начатая по предложению энтузиастов этого языка в ЛГУ, имеет большое практическое значение. Реализация языка Рефал для МВК "Эльбрус" позволила снять многие ограничения по времени и по памяти, с которыми постоянно сталкиваются пользователи Рефала на других ЭВМ.

Разработка системы Рефал-Эльбрус и ее принципов выполнена А.А. Кубенским, М.В. Дмитриевой и В.А. Лаврищевой.

Элементы языка Рефал. Программа на языке Рефал представляет собой последовательность *предложений* (правил вывода), имеющих вид:

$$K/D/E = F.$$

где K – *начальный символ конкретизации* (признак начала предложения), D – *детерминатив* (идентификатор, играющий роль имени функции),

E – текстовое выражение (аргумент); F – также текстовое выражение, возможно, пустое (результат конкретизации). Если F пусто, точка опускается. Символы "=" и "." играют роль *конечных символов конкретизации:* символ "=" ограничивает справа выражение E, символ "." – выражение F. Смысл предложения: найти в поле зрения первое слева внутреннее выражение Z, заключенное в скобки K и . , выполнить синтаксическое отождествление (унификацию) Z с E и в случае успеха осуществить замену выражения KZ . поля зрения на выражение F', полученное из F подстановкой значений переменных, полученных в результате отождествления. Предложение (выражения E и F) может содержать *переменные* вида SX (*символ*), WX (*терм*) или EX (*выражение*), где S, W и E – тип переменной, X – ее идентификатор. Переменная X может принимать в качестве значения текст, имеющий соответственно вид символа, терма или любого выражения. При успешном синтаксическом отождествлении определяются значения переменных, входящих в выражение (подобно тому как происходит неявное присваивание при сопоставлении с образцом в языке Снобол-4), и эти же значения подставляются в выражение F, после чего сопоставленное выражение KZ. в поле замещается результатом подстановки (F'). При удачном отождествлении выполняется переход к следующему предложению рефал-программы, имеющему тот же детерминатив D, либо аварийная остановка рефал-машины, если такого предложения нет. Нормальная остановка рефал-машины выполняется, если перед очередным шагом в поле зрения нет выражений для конкретизации, имеющих вид KZ. . При запуске рефал-машины ей передаются: текст программы (начальное состояние поля памяти); исходный текст (начальное состояние поля зрения), заключенный в скобки K .; начальный детерминатив. Из дополнительных возможностей языка Рефал отметим "*закапывание*" и "*выкапывание*" (удаление и возвращение подвыражений поля зрения по принципу стека).

При составлении рефал-программы рекомендуется упорядочивать последовательность предложений от более частных к более общим, что соответствует естественному процессу индуктивного вывода.

Традиционный пример простой рефал-программы – переворачивание слова:

K* / *П* / *E 1 ***S*** 2 = ***S*** 2/***П***/***E*** 1.

***K* / *П* /** =   .

В применении к строке "НОФЕЛЕТ" в поле зрения эта программа работает следующим образом (указаны только состояния поля зрения после каждого шага вывода):

K / П / НОФЕЛЕТ .→ Т/ K / П/ НОФЕЛЕ .→ ТЕ K /П/ НОФЕЛ .→
→ТЕЛ K /П/ НОФЕ . → ... → ТЕЛЕФОН K /П/ . → ТЕЛЕФОН

Система Рефал-Эльбрус [30]. Эта система предназначена для интерпретации и отладки рефал-программ. Система может использоваться как независимая система программирования либо как процедура преобразования текста, вызываемая, например, из программы на языке Эль-76.

Если Рефал-система используется как независимая, то ее вызов должен иметь вид

исп ксн // символ // рефал (прог, дан, начдет, режимы)

где *ксн//символ//рефал* – стандартное внешнее имя файла объектного кода рефал-системы; *прог* – текст программы (если *прог* = э, то текст вводится с терминала); *дан* – исходные данные (начальное состояние поля зрения); *начдет* – начальный детерминатив; *режимы* – режимы работы рефал-системы. Исходные данные могут быть заданы либо строкой: стр8 "НОФЕЛЕТ", либо текстовым файлом, содержимое которого рассматривается как одна строка. Начальный детерминатив указывается строкой или набором в форме: "*П*" или "/*П*/". Возможно указание начального детерминатива непосредственно в строке исходных данных: стр8 "*К*/*П*/НОФЕЛЕТ". Режимы (необязательный параметр) задаются изображением строки, например; стр8 "¤*л*: + ¤*п*: + ¤*т*: +" – интерпретация с предварительной распечаткой текста программы и начального состояния поля зрения (*л*); выдача протокола в стандартный АЦПУ-файл (*п*); трассировка работы рефал-машины (*т*).

Рефал-система может использоваться для трансляции текста программы во внутреннее представление. Если параметры имеют вид (*текст*, 0, *внутр*), где *внутр* – некоторый файл на барабане или диске, то система записывает в файл *внутр* внутреннее представление рефал-программы для интерпретации. Далее файл *внутр* (вместо исходного текста) может быть задан первым параметром при запуске программы на интерпретацию.

При запуске рефал-системы в диалоговом режиме обязателен лишь первый параметр; значения остальных при необходимости запрашивается системой. Например:

исп ксн//символ//рефал (э)

* система Рефал-Эльбрус. версия . . . дата . . . *
введите текст программы:

>*К* /*П* / *Е* 1 *S* 2 = *S* 2 /*П*/ *Е* 1.

> *К* /*П*/ =

@

введите исходные данные:

>НОФЕЛЕТ

введите начальный детерминатив:

>*П*

результат конкретизации:

ТЕЛЕФОН

конец работы рефал-системы

>

Если система Рефал-Эльбрус запускается из другой программы для преобразованного текста, то следует использовать другое стандартное внешнее имя точки входа в систему – *ксн//символ//брефал*. В качестве результата выдается преобразованный текст в виде строки. Например:

начало

конст *рефал* = *прог* (*ксн//символ//брефал*),

```
        рефпрог = тфайл * конреф *
                          K/верни/E1S2 = S2·K/верни/E1.
                          K/верни/ =
                     конреф,
        дет = "верни",
        печ = читатрзадачи (виртвыв);
    запф (печ, рефал (рефпрог, стр8 "нофелет", дет))
  конец
```

В тексте рефал-программы можно использовать библиотечные карты в традиционной форме: ¤ *биб //текст*. В этом случае рефал-системе пятым параметром должен быть передан справочник – основа внешних имен.

4.4. Диалоговая система ДИАШАГ

Диалоговая система программирования с пошаговой обработкой программ ДИАШАГ [59] создана в 1985 г. В.Н. Рябининым и В.Ю. Самойловым. Им принадлежат принципы ее построения, язык управления и реализация системы.

Пошаговые трансляторы – общепризнанное удобное средство для разработки и отладки программ в диалоговом режиме. Основное отличие их от обычных ("пакетных") трансляторов в том, что текст программы для пошагового транслятора может вводиться и корректироваться *шагами* – фрагментами программы, разделение на которые определяется программистом. "Пакетные" трансляторы обеспечивают лишь полную трансляцию программы или перетрансляцию ее отдельных процедур (см. гл. 2).

Каждый шаг программы имеет свой *номер*, который задается при вводе или коррекции шага. Последняя вызывает его автоматическую перетрансляцию. Выполнение программы также может осуществляться шагами, даже в том случае, если программа введена еще не полностью. Пошаговые трансляторы близки весьма популярным в настоящее время синтаксически-ориентированным редакторам. Первыми пошаговыми системами программирования, по-видимому, являются реализации языка Бейсик. В этом языке с каждым оператором связан номер, по которому может выполняться его коррекция в диалоге. Принцип пошаговой трансляции требует совершенно иной организации транслятора, включающей значительные элементы интерпретации.

Система ДИАШАГ выделяется среди пошаговых систем программирования удобными возможностями языка управления и эффективной реализацией (каждый шаг компилируется). Входным языком первой системы ДИАШАГ является Алгол-60. Версия входного языка согласована со входным языком основной системы программирования Алгол-60 для МВК "Эльбрус", что дает возможность отлаживать алгол-программы в системе ДИАШАГ, а пакетный алгол-транслятор использовать для компиляции отлаженных программ.

Шаг программы может состоять из одного или нескольких описаний либо одного или нескольких операторов. Для удобства записи программ разрешено использовать в качестве шага условие оператора *if*, часть этого оператора до символа **else**, заголовок процедуры или цикла. В качестве на-

чального *маркера шага* используется символ ¤ (денежный знак). После маркера шага следует его *номер*, который для удобства коррекции программы может быть не только *целым*, но и *вещественным числом*. Служебные слова выделяются либо апострофами (*'begin'*), либо подчеркиванием и пробелом (_ *begin*) и могут сокращаться до минимального числа символов, при котором это не приводит к двусмысленности (например, *'b'* или *'be'*).

Пример ввода программы по шагам:

¤ 10 *'be' 'in' a*;
¤ 20 *ввод* (*a*);
¤ 30 *печать* (*a*)
¤ 40 *'en'*

Здесь *ввод* и *печать* – стандартные процедуры ввода с терминала и вывода на терминал.

Язык управления системой состоит из набора директив. Каждая директива набирается в отдельной строке и состоит из маркера директивы %, ее названия и параметров. Основные директивы:

1. *%уст тр* – установка режима автоматической трансляции каждого введенного шага.

2. *%отм тр* – отмена режима автоматической трансляции.

3. *%тр* – трансляция всех введенных шагов (при отключенном режиме *тр*).

4. *%текст* 5–20; – вывод текста заданного диапазона шагов на терминал.

5. *%вып* 1–10, 25–30; – выполнение заданных шагов. Если диапазон не указан, то выполняется вся программа. *%вып* * – выполнение текущего шага. Выполнение шагов с *трассировкой* задается ключом *т*: *%вып* 1–10 *т*;

6. *%вр* – выдача текущего времени.

7. *%ст* – конец работы.

8. *%зап текст* 0 + 1000; – запись текущей программы в файл *текст* базового контейнера на диске, переданного системе в качестве параметра. Каждый шаг записывается в виде отдельной строки. Директива записи позволяет осуществить переход от режима отладки в системе ДИАШАГ в режим трансляции пакетным алгол-транслятором. Если указан ключ *: *%зап * текст* 0 + 1000, то для записи текста создается временный файл на барабане, который уничтожается в конце сеанса.

9. *%чит текст* 0 + 1000 *п*; – чтение текста из файла. Каждая строка текста преобразуется в шаг, к ней добавляются маркер шага и номер в соответствии с заданным началом (0) и приращением (1000). Ключ *п* задает режим *построчного просмотра* текста: пустая посылка в ответ на приглашение "?" – прием строки, ответ "–" – отклонение, ответ в виде нового варианта строки – модификация. Если ключ *п* опущен, то чтение выполняется автоматически. Директива чтения позволяет ввести в систему новый текст.

10. *Прямой шаг* – текст на Алголе-60 без номера шага: *печать* (*x, y*); Прямой шаг – дополнительное средство отладки, позволяющее вывести значения переменных и выполнить другие отладочные действия, которые

не запоминаются в виде шагов. Прямые шаги немедленно транслируются и исполняются.

11. *%ред ф1 ф2* – вызов текстового редактора из системы ДИАШАГ. *ф1, ф2* – имена архивных и временных файлов (временный помечается символом "*").

Обращение к системе ДИАШАГ осуществляется директивой

исп ксн//дсп//алгол (э, *реж*)

или

исп ксн//дсп//алгол (э, *реж, спр*)

где э – ссылка на терминальный файл; *реж* – режим выдачи протокола (1 – ведется на АЦПУ, 0 – не ведется); *спр* – архивный справочник. Третий параметр обязателен, только если используются архивные файлы.

Пример сеанса разработки и отладки программы в системе ДИАШАГ:

```
>кн ксн//дсп
>исп//алгол (э, 1)
  <* диашаг, версия от 10.09.87 *>
= %отм тр
= ¤ 10 'begin' 'integer'n;
= ¤ 20 ввод (n);
= ¤ 30 'begin' 'array'a [1:n];
= ¤ 40          'integer'i; 'real'm;
= ¤ 50          'boolean'b;
= ¤ 60      ввод (a); печать (a); b := 'true';
= ¤ 70      'for'm := m  'while' b 'do'
= ¤ 80      'for'i := 1     'step' 1 'until'n–1'do'
= ¤ 90      'if' a [i] > a[i + 1] 'then'
= ¤ 100     'begin'
= ¤ 110          m := a [i]; a [i] := a[i + 1]; a[i + 1] := m;
= ¤ 120     'end';
= ¤ 130 печать ('результат =', a)
= ¤ 140 'end'
= %текст 70–80;
70  'for'm := m 'while' b 'do'
80  'for' i := 1 'step' 1 'until' n–1 'do'
= ¤ 75  'begin' b := 'false';
= % текст 110–120
  110  m := a[i]; a[i] := a[i + 1]; a[i + 1] := m;
  120  'end';
= ¤ 115  'end'
= % текст
(выдается исправленный текст программы)
= %тр
  трансляция введенных шагов закончена
= %вып
n = 5
a = 5.0 4.0 3.0 2.0 1.0
   5.0 4.0 3.0 2.0 1.0
```

```
результат = 4.0 3.0 2.0 1.0 5.0
= ¤ печать (в)
   false
= %вып 20–60
   n = 3
a = 3.0 2.0 1.0
= %вып 70–120 т;
   выполнили шаг 70, переходим в шаг 75
   выполнили шаг 75, переходим в шаг 80
   выполнили шаг 80, переходим в шаг 90
   выполнили шаг 90, переходим в шаг 100
   выполнили шаг 100, переходим в шаг 110
   выполнили шаг 110, переходим в шаг 115
   выполнили шаг 115, переходим в шаг 70
   выполнили шаг 70, переходим в шаг 120
   останов в шаге 120
=  ¤печать (в) ; печать (а);
   false 2.0 1.0 3.0
= %текст 110–120
   110      m := a[i] ; a[i]  := a[i + 1] ; a[i + 1]  := m;
   115      'end'
   120      'end';
=  ¤111     b := 'true';
= %тр
   трансляция введенных шагов закончена
= %вып
n = 3
a = 3.0 2.0 1.0
результат = 1.0 2.0 3.0
= %ст
   <* конец работы системы диашаг *>
   >печ ацпу
```

В данном сеансе вводится, корректируется и отлаживается программа сортировки массива. Программист допускает две ошибки, связанные с использованием логической переменной *в*. Первую из них (не выполнена инициализация *в* перед началом просмотра массива *а*) программист находит и исправляет визуально, еще в процессе ввода программы (до ее трансляции), добавляя шаги 75 и 115. Вторая (переменной *в* не присвоено значение *'true'* после перестановки элементов, не удовлетворяющих отношению упорядочения по возрастанию) обнаруживается только в ходе отладки программы. Введенная программа транслируется и исполняется. Для ввода числа *n* и массива *а* система выдает приглашение в виде: *n* = (*а* =). Обнаружив неверный результат, программист выполняет прямой шаг – выводит значение *в* (оно оказывается ложным). После этого начинается диалоговая отладка программы. Сначала вводятся более простые исходные данные (*n* = 3; массив *а* из трех чисел); для этого программа частично выполняется (от шага 20 до шага 60). Затем запускается выполнение основной части программы – циклов (шаги 70–120) с трассировкой, и программист

обнаруживает, что внутренний цикл выполняется всего один раз. После этого вновь выполняется прямой шаг (выводятся значения *в* и *а*), и ошибка обнаруживается. Программист выводит содержимое шагов 110–120 и исправляет программу (добавляет шаг 111). Наконец, директивой *%тр* введенный шаг 111 транслируется, и программа исполняется полностью (результат верный). После выхода из системы протокол распечатывается на АЦПУ (режим вывода протокола на АЦПУ установлен вторым параметром при запуске системы).

Как видно из примера, разработка и отладка программы с помощью пошагового транслятора весьма удобна: обеспечивается оперативность "вмешательства" в процессе исполнения программы, ее частичное выполнение, коррекция, трассировка. Отлаженная программа может быть записана в архив. Столь гибких возможностей отладки программ в диалоге не имеет в системе "Эльбрус" никакая из эксплуатационных систем программирования.

4.5. Диалоговая система Форт-Эльбрус

Язык программирования Форт [82] в последнее время популярен среди системных программистов, главным образом среди разработчиков программного обеспечения мини- и микроЭВМ. Он рассчитан на работу в диалоговом режиме, очень прост, обладает легкой расширяемостью и значительной гибкостью. Программирование на языке Форт состоит в определении и использовании (интерпретации) *команд*, выполняющих последовательности действий и обозначаемых идентификаторами (словами). Группа определений слов объединяется в *словарь*, который может использоваться в ходе диалога и запоминаться во внешней памяти. Словарь можно рассматривать как определение языка для разработки программ в некоторой предметной области. При запуске форт-системы она настроена на стандартный словарь, содержащий базовые операции языка Форт. Над стандартным словарем программист может надстраивать совокупность пользовательских словарей; поиск определения слова для его интерпретации выполняется начиная с последнего введенного словаря.

В языке Форт для хранения промежуточных результатов используется *стек*. Большинство стандартных операций берут из стека операнды и записывают в стек результат. Поэтому программа на Форте представляет собой обратную польскую запись выражения, т.е. в некоторых отношениях близка программе в кодах МВК "Эльбрус". Стековая организация вычислений значительно упрощает реализацию форт-системы, но резко снижает наглядность и надежность программ. Тем не менее, несмотря на почти полное отсутствие защиты от ошибок, язык Форт активно используется благодаря своему основному преимуществу – простоте средств создания собственных диалектов языка и их применения.

П р и м е р определения слова на языке Форт: возведение в натуральную степень (*x* ** *n*):

: *power* 1 *swap* 0 *do over* * *loop swap drop*;

Здесь: 1, 0 – загрузка в стек целых констант, *swap* – перестановка двух верхних ячеек стека, *over* – дублирование слова стека, лежащего ниже

вершины, * – арифметическая операция умножения, *drop* – вычеркивание вершины стека. Пара слов *do* и *loop* предназначена для организации циклов. Команда, обозначенная словом *do*, имеет два операнда в стеке – $n + 1$ и m, где n – предельное значение индекса, m – начальное значение; слово *do* формирует индекс цикла и записывает его в стек возвратов. Команда *loop* осуществляет продолжение или завершение цикла (число повторений равно $n - m - 1$).

Команда *power* имеет два аргумента: число x и показатель степени n, которые должны быть загружены в стек. Пример использования команды *power*:

5 3 *power*.

На экран будет выведено значение 125 (. – печать значения из вершины стека и вычеркивание его из стека).

Система Форт-Эльбрус. Эта система создавалась под влиянием работ сотрудников ЛГУ по созданию программного обеспечения дисплейного комплекса ЕС-7970 на базе форт-системы [38], в ходе которых был разработан большой объем полезных программ на этом языке. Разработка системы Форт-Эльбрус имела и чисто исследовательские цели как опыт реализации на МВК "Эльбрус" языка с нетрадиционной семантикой, уровень которого близок к ассемблерному.

Принципы разработки, структура и реализация системы Форт-Эльбрус принадлежат А.Е. Соловьеву.

Вызов системы Форт-Эльбрус выполняется директивой *исп* стандартного диалога со следующим набором фактических параметров:

терминальный файл;

файл с начальным словарем;

файл, используемый в качестве входного потока (необязательный параметр).

Входным языком системы Форт-Эльбрус является расширение стандартного языка FORTH-83 [82].

При входе в форт-систему выполняется разметка экрана на следующие поля:

– *поле системы* – строки 1–2: для ввода текста по директиве *expect* [82];

– *поле вывода* – строки 3–22: для вывода результатов работы форт-программы и сообщений об ошибках;

– *поле идентификации* – строка 23: как правило (при вводе очередной строки текста программы), содержит стандартный текст:

МВК "Эльбрус". система FORTH-83·

– *поле ввода* – строка 24: строка для ввода текста форт-программы (при входе в систему курсор автоматически устанавливается на эту строку).

Особенности реализации. Язык Форт на традиционных ЭВМ используется как язык, близкий к языку ассемблера, так как стандартные слова языка Форт обеспечивают доступ к реальным адресам, адресную арифметику и другие ассемблерные возможности, характерные для ЭВМ обычной архитектуры. Реализация отдельных команд в большинстве форт-систем может быть выполнена на языке ассемблера. На МВК "Эльбрус" для воспроизведения этих возможностей было смоделировано традицион-

ное адресное пространство в виде вектора переменной длины. Другая важная особенность реализации – использование средств физического обмена с терминалом ЕС-7920, на основе которого реализована описанная выше многооконная структура экрана. Система Форт-Эльбрус совместима по входному языку со стандартом FORTH-83.

4.6. Система Модула-2-Эльбрус

Модула-2 [88] – язык модульного программирования, созданный Н. Виртом на основе языка Паскаль. Язык Модула-2, как и язык Ада, – самый популярный в настоящее время язык программирования. От языка Ада его отличает компактность, присущая и его предшественнику – языку Паскаль. В Модуле-2 в основном сохранены черты языка Паскаль. Наиболее существенное дополнение – *средства модульного программирования* (*модули, определяющие модули, реализующие модули, экспорт* и *импорт* величин между модулями). Именно таких средств не хватало в языке Паскаль (п. 2.1). В отличие от языка Паскаль в Модуле-2 отсутствуют упакованные типы, что упрощает проблемы представления данных. Введен новый стандартный тип *cardinal* (натуральное число), процедурные типы (*proc*) и типы *word* и *address* для ограниченной динамики и работы с адресной информацией.

Язык Модула-2 ориентирован на мини- и микроЭВМ (авторская реализация выполнена на *PDP*-11). Эта ориентация выражается прежде всего в ограничениях на средства абстракции данных и управления модулями, а также во введении в язык понятий, характерных для этого класса ЭВМ (приоритет модуля).

Модуль и описываемые в нем величины существуют в единственном экземпляре. Модуль не имеет статических параметров, как в языке Клу (см. гл. 3). *Определение модуля* содержит *список импорта*, в котором указаны доступные в нем величины из других модулей, и *список экспорта*, содержащий имена некоторых величин, описанных внутри модуля. Все информационные связи между независимыми модулями (отдельными единицами компиляции) фиксированы в списках экспорта-импорта. Для пошаговой разработки программ введена возможность раздельного описания и раздельной компиляции интерфейса модуля (**definition module**), содержащего предварительные описания экспортируемых величин, и реализации модуля (**implementation module**), в которой даны окончательные описания всех объектов интерфейса и описания скрытых объектов модуля. Как и в языке Клу, предполагается, что перед исполнением программы выполняется ее *сборка* из используемых в ней модулей.

П р и м е р. Определение кольцевого буфера (очереди). Чтение возможно из одного конца буфера, запись выполняется в другой конец.

```
module buffer;
import underflow, overflow;
export readbuf, writebuf;
const buflen = 256;
type indbuf = [0 . . . buffen – 1];
var lbuf, rbuf: indbuf; n: integer;
    buf: array indbur of real;
```

```
procedure readbuf( ): real;
var res: real;
begin if n = 0 then underflow ( )
else res := buf [lbuf]; n := n – 1;
   lbuf := (lbuf + 1) mod buflen; return res
  end
end readbuf;
procedure writebuf (x : real);
  begin if n = buflen then overflow ( )
     else  n := n + 1; buf [rbuf] := x;
           rbuf := (rbuf + 1) mod buflen
  end
end writebuf;
begin lbuf := 0; rbuf := 0; n := 0
end buffer
```

Модуль *buffer* импортирует процедуры *underflow* и *overflow*, вызываемые соответственно при исчерпании и переполнении буфера, и экспортирует процедуры (операции) *readbuf* и *writebuf* – чтение из буфера и запись в буфер. В модуле описано *конкретное представление* буфера в виде совокупности констант, типов и переменных. В разделе операторов модуля *buffer* приведена инициализация конкретного представления.

Система Модула-2-Эльбрус создается на основе системы Паскаль-Эльбрус (см. гл. 2). Цели разработки – реализация популярного языка программирования и эксперимент по разработке транслятора на основе другого транслятора с близкого по синтаксису и семантике языка с применением ТИП-технологии программирования (см. гл. 5). Метод разработки автоматически обеспечивает унификацию формы обращения к системе и управляющих карт с системой Паскаль-Эльбрус. Входной язык – расширение описания [88]. По аналогии с системой Паскаль-Эльбрус введены дополнительные стандартные типы *shortint* (целое 32), *shortreal* (вещественное 32) и *longreal* (вещественное 128). Единицами компиляции в системе являются полное определение модуля (*module*), определение интерфейса модуля (*definition module*) и реализации модуля (*implementation module*). Компиляция интерфейса модуля должна предшествовать компиляции его реализации. Определение модуля транслируется в расширенный файл объектного кода модульного объекта, который записывается в файл, переданный третьим параметром компилятору. В системе Модула-2-Эльбрус, как и в системе Клу-Эльбрус, для хранения информации о модулях используется общий архив и стандартная структура расширенного файла объектного кода.

Как и в системе Клу-Эльбрус, реализованы два режима сборки программы – статический и динамический. Однако, в отличие от языка Клу, сборка в Модула-2 выполняется гораздо проще, так как не требует каких-либо конкретизаций. При динамической сборке в ходе исполнения программы происходит обмен информацией между экземплярами модульных объектов. При статической сборке все модули объединяются в одну программу. Режим статической сборки задается при вызове системы Мо-

дула-2-Эльбрус управляющей картой:

¤ сборка //m,

где //m – внешнее имя файла объектного кода головного модуля. Специального режима трансляции для модулей (как в клу-системе) при статической сборке не требуется, так как объектный код модулей не конкретизируется. При исполнении программе передаются фактические параметры головного модуля (как правило, файлы ввода-вывода).

ГЛАВА 5

ТИП-ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

По мере накопления опыта разработки больших программ в некоторой предметной области, в каждом коллективе программистов складываются определенные технологические принципы программирования. В большинстве случаев они носят неформальный характер и применяются только внутри данного коллектива. На основе их программной поддержки, организационного обеспечения, документирования и отработанной методики применения создаются *технологии программирования*, которые могут быть переданы для использования другим группам разработчиков. В настоящее время в мире насчитывается около 200, а в СССР – около 50 технологий программирования.

В данной главе излагаются принципы *ТИП-технологии программирования* [65, 68], которые сложились и были применены в процессе разработки систем программирования для МВК "Эльбрус", описанных в гл. 2–4. ТИП-технология предназначена для создания программ, связанных с обработкой сложных структур данных. Ее характерные свойства – развитие концепций АТД и интерфейса. Сформулирован общий подход к технологиям программирования, даны краткий обзор основных отечественных технологий, мотивировка ТИП-технологии и ее сравнение с другими технологиями. Рассмотрены основные технологические единицы, процесс проектирования и разработки программ по ТИП-технологии, требования к языковым средствам для ее применения, принципы программирования по ТИП-технологии на языках Эль-76, Паскаль, Клу и Модула-2 в системе "Эльбрус". Проанализирован опыт и даны рекомендации по применению ТИП-технологии при разработке трансляторов для МВК "Эльбрус". В заключение рассмотрена организация программной поддержки ТИП-технологии – технологического комплекса ТИП для разработки программ по ТИП-технологии программирования в диалоговом режиме.

Одной из целей изложения ТИП-технологии в этой книге является анализ и обобщение десятилетнего опыта разработки больших системных программ на языке Эль-76. Хотелось бы надеяться, что содержание главы заинтересует специалистов по различным вычислительным системам и будет способствовать распространению ТИП-технологии не только на МВК "Эльбрус", но и на других типах ЭВМ.

5.1. Общий подход к технологиям программирования

В современных технологиях программирования можно выделить три основные составные части:

– *программные средства* поддержки этапов жизненного цикла разработки программ (редакторы, трансляторы, средства построения трансляторов, отладчики, комплексаторы, верификаторы, документаторы, системы поддержки версий, технологические комплексы и др.);

– *организационные средства* – поддержки процесса создания программ (формы разделения труда в коллективе программистов, графики работ, методики оценки принимаемых решений, средства контроля выполнения работ);

– *концептуальные средства* поддержки стиля проектирования и разработки программ (Р-схемы, абстрактные типы данных, вычислительные модели, интерфейсы и т.д.).

Каждая из этих частей в равной степени необходима для успешного применения технологии программирования.

Рассмотрим с этой точки зрения некоторые известные отечественные технологии программирования. Эта же точка зрения принята и при описании особенностей ТИП-технологии.

5.2. Обзор технологий программирования

Исторически первой отечественной технологией программирования, по-видимому, является *технология вертикального слоения* [75]. Это система принципов разработки больших программных систем, наиболее важными элементами которой являются две концепции:

– *вертикальный слой* – рассредоточенная совокупность элементов программной системы, реализующих какую-либо ее отдельную возможность (функцию); по принятой классификации, эта концепция относится к третьей группе и является основным концептуальным средством этой технологии;

– *вариантная сеть* – система для количественной оценки принимаемых проектных решений на основе численных весов различных вариантов. Вариантную сеть можно отнести к организационным средствам. Созданы также и программные средства поддержки этой технологии.

Наиболее разработанной во всех трех рассмотренных выше аспектах является *Р-технология программирования* [14]. Ее концептуальными средствами являются *Р-схемы*, имеющие наглядное графическое представление в виде нагруженных ориентированных графов. Р-технология основана на графическом стиле программирования, реализует принцип управления программой "от данных" и, по-видимому, является наиболее подходящей в задачах, связанных с обработкой последовательных потоков данных (текстов, сообщений и т.д.). Р-технология имеет развитые программные средства поддержки – технологический комплекс РТК, реализованный на всех распространенных типах ЭВМ, в котором имеются элементы поддержки организационных средств Р-технологии. Р-схемы удобны для изображения управляющей структуры модулей, зависящей от структуры обрабатываемых данных, но, по-видимому, мало подходят для выражения

структуры интерфейса технологических пакетов (абстрактных типов данных) и для описания рекурсивных структур данных и управления.

П Р И З — т е х н о л о г и я п р о г р а м м и р о в а н и я [72]. Эта технология обеспечивает высокий концептуальный уровень разработки. Ее концептуальными средствами являются *вычислительные модели* — конструкции базового языка Утопист для описания предметной области, ее объектов и соотношений между ними. Технологическое окружение ПРИЗ-технологии состоит из инструментальных систем ПРИЗ и ДИМО. Инструментальная система ПРИЗ выполняет структурный синтез модулей вычисления выходных величин по заданной модели и входным величинам. Другим инструментальным средством ПРИЗ-технологии является система ДИМО, предназначенная для поддержки разработки диалоговых программ на основе *фреймов диалоговых ситуаций*, которые представляют собой другой вид концептуальных средств ПРИЗ-технологии. Организационная часть ПРИЗ-технологии поддерживается системой ПРОЕКТ для введения базы данных проекта.

Ф о р т - т е х н о л о г и я п р о г р а м м и р о в а н и я [38]. Данная технология, применяемая при разработке программного обеспечения терминальных станций ЕС-7970, основана на системе программирования Форт, которая является базовым программным средством поддержки этой технологии. Концептуальные средства форт-технологии — механизм слов и словарей, который можно рассматривать как гибкое средство для хранения, модификации и использования системы понятий (слов) для разработки программ в некоторой области. Однако в языке Форт семантика этих понятий может быть выражена лишь на языке абстрактной стековой машины или на языке ассемблера. Аргументы и результат фортовской команды хранятся в стеке. Интерфейс модуля может быть описан лишь в виде комментария, поэтому система не имеет почти никаких средств контроля правильности интерфейса при обращении к модулю. Однако весьма важным свойством форт-системы с точки зрения технологии программирования является поддержка стиля программирования снизу вверх небольшими модулями, с постепенным накоплением полезных модулей в словарях.

Т е х н о л о г и я С У П Е Р Ф О Р М А Т. Эта технология, разрабатываемая под руководством А.Я. Диковского, базируется на использовании специализированных программных средств, составляющих технологический комплекс СУПЕРФОРМАТ. Этот комплекс предоставляет пользователю спектр связанных в единую систему языков для систематического выполнения всех этапов разработки программы: язык спецификации, язык проектирования, язык реализации. Языки спецификации и проектирования содержат необходимые концептуальные средства. Язык реализации построен на основе современных языков Ада и Модула-2. Идея такого интегрированного технологического комплекса весьма перспективна. Комплекс СУПЕРФОРМАТ является замкнутой системой: для его применения пользователь должен изучить все перечисленные языки и работать только на них, а не на привычных ему языках (например, на Фортране или Паскале).

С б о р о ч н о е п р о г р а м м и р о в а н и е. Эта концепция развивается в работах Г.С. Цейтина. Ее основной принцип состоит в том, что большие программы должны строиться путем сборки из независимых модулей.

При этом каждый модуль рассматривается как элементарная неделимая единица алгоритмического знания, реализующая некоторую независимую идею, т.е. в технологии сборочного программирования концептуальными средствами являются сами модули, составляющие базу алгоритмических знаний в некоторой области. Программа конструируется из модулей с помощью макрогенерации и смешанных вычислений. В настоящее время разрабатываются программные средства поддержки этой технологии.

ПРОМЕТЕЙ-технология. Одной из первых отечественных промышленных технологий массового производства программной продукции является промышленная технология разработки программных средств для встраиваемых микро-ЭВМ (технология ПРОМЕТЕЙ [47]). Эта технология основана на системе взаимосвязанных языков проектирования, поддерживающих все этапы технологического процесса (жизненного цикла программного комплекса): язык спецификации требований, язык настройки на конкретные встроенные ЭВМ, языки разработки программ (автокоды, макроязыки, языки высокого уровня), языки автономной и комплексной отладки. Эта система языков дает удобные концептуальные средства для описания требований к комплексу программ, его функциональной и логической структуры, взаимосвязей между модулями, систематизации процесса разработки программ. Инструментальные программные средства ПРОМЕТЕЙ-технологии образуют пять систем: организационную, автоматизации программирования, автоматизации отладки, изготовления программного изделия и информационную. Работа программиста осуществляется в диалоговом режиме и поддерживается базой данных проектирования на всех этапах технологического процесса. ПРОМЕТЕЙ-технология имеет программные средства поддержки – инструментальные системы ЯУЗА, ТЕМП (БЭСМ-6), РУЗА, ПРОТВА (ЕС ЭВМ), ПРА (микроЭВМ). Характерной чертой этой технологии является интегрированный подход к процессу разработки программ для встраиваемых ЭВМ.

Технология проектирования структуризованных программ на основе псевдокода [28]. Псевдокод – это частично формализованный язык для наглядного текстового представления схем алгоритмов и программ, разрабатываемых по принципам структурного программирования. Псевдокод можно рассматривать как совокупность концептуальных средств для проектирования программ. Он используется для пошаговой детализации фрагментов алгоритма, для формулирования заданий на программирование; для описания логики работы программ с целью их документирования. Псевдокод не поддержан традиционными программными средствами (трансляторами), так как носит неформальный характер. Конструкции псевдокода: определения и вызовы функций, обозначения условий, схемы управляющих конструкций структурного программирования. По мнению автора этого подхода [28], проектирование программ на псевдокоде более удобно, чем использование блок-схем, так как текст на псевдокоде легче, чем блок-схемы, вводить и модифицировать. Автор отмечает и ограничения псевдокода: он не содержит средств для описания используемых структур данных, поэтому программу на псевдокоде необходимо дополнять специальными справочниками, содержащими сведения об организации данных.

Типовая технология разработки программ для МВК "Эльбрус" [53]. Эта технология основана на аппаратной поддержке языков высокого уровня в системе "Эльбрус". Она обеспечивает разработку, отладку и сопровождение программы, написанной на одном или нескольких языках высокого уровня. Отладка выполняется в терминах исходного языка. Основными элементами данной технологии являются: базовая система программирования Эль-76; концепция модульного объекта (п. 1.31), имеющая системную и языковую поддержку; многоязыковые компоненты ОСПО: система динамической диагностики, комплексатор, символьный отладчик, система получения словаря идентификаторов программы. Разрабатываются инструментальные средства управления программными конфигурациями и версиями и средства организации работы коллектива программистов. Таким образом, имеются удобные программные средства и создаются организационные средства поддержки этой технологии. В соответствии с общей тенденцией разработки аппаратных и программных компонент МВК "Эльбрус" программисту не навязывается определенная дисциплина, стиль программирования, режим работы, язык и т.п., а предоставляется набор базовых средств, связанных общей методологической, аппаратной и программной основой, объединяемой под общим названием – ориентация на языки высокого уровня. На основе типовой технологии разработки программ для МВК "Эльбрус" могут создаваться различные конкретные технологии программирования.

5.3. Принципы ТИП-технологии

5.3.1. Мотивировка. Одним из распространенных современных подходов к программированию является *объектно-ориентированный подход*, при котором программа рассматривается как последовательность операций над одним или несколькими *объектами*. Под объектом понимается некоторая структура данных либо понятие, устройство или процесс, который выражен в программе в виде совокупности характеризующих его данных. Поведение объекта характеризуется совокупностью *атрибутов* (свойств). Сложные объекты (таблицы, массивы, списки, деревья и др.) могут состоять из нескольких *компонент* (элементов, узлов), каждая из которых может сама иметь собственные атрибуты. В любой программе можно выделить следующие основные классы действий над объектами:

1) *конкретное представление* (описание) – выражение структуры и атрибутов объекта через встроенные или определяемые типы данных инструментального языка программирования;

2) *генерация* – создание объекта или его компонент (генерация может выполняться частично или полностью уже при исполнении описания объекта);

3) *модификация* – изменение состояния объекта;

4) *доступ* – обращение к атрибутам объекта и их анализ;

5) *вывод* – преобразование информации об объекте и его состоянии в форму, наиболее удобную для восприятия человеком или программой дальнейшей обработки объекта.

Разумеется, данная классификация для некоторых классов объектов достаточно условна: например, для файлов (при буферизованном обмене)

действия, связанные с доступом, модификацией и выводом, неотделимы друг от друга.

Большинство языков программирования содержат конструкции для работы с элементарными (встроенными) объектами. Например, в языке Паскаль простейшая разновидность объекта – это переменная предопределенного типа, для которой в язык встроены все четыре рассмотренных класса действий:

1) конкретное представление и генерация – описание переменной: **var** *i* : *integer*;

2) доступ – использование переменной: *i* + 1;

3) модификация – присваивание переменной: *i* := 0 или ввод *read* (*i*);

4) вывод – предопределенная процедура вывода: *write* (*i*).

Операции над сложными объектами, определяемыми программистом, могут быть выражены в виде описаний процедур. Однако размер этих процедур может оказаться весьма малым: например, реализация операции может представлять собой выборку или изменение значения компоненты переменной сложного типа, вычисление выражения и т.п. В таком случае реализация операций в виде процедур может ухудшить эффективность программы. В связи с этим сложилась практика программирования операций над сложными объектами в явном виде, в терминах конкретного представления, которая приводит к тому, что в программе трудно различимы границы фрагментов текста, относящихся к разным операциям.

Пример ненадежного программирования на Эль-76: работа со списками.

```
конст список = лок вект [100] ф64; ф64 ук := 1, тек;
список ук :=. ([63:32]:1, [31:32]:ук + 1);
список [ук + 1] :=. ([63:32]:2);
ук := * + 2; % создан список (1 2)
тек := 1; % поиск элемента, равного 2
до конец цикл
    если тек = 0 то конец!
    инес список [тек] · [63:32] = 2 то конец!
    иначе тек := список [тек] · [31:32] все повторить
```

В работе [35] это свойство программ называется ***временной связностью модулей*** и рассматривается как пример плохого стиля и слабой надежности программы.

В силу особенностей интеллектуальной деятельности человека такой стиль программирования приводит к ошибкам; наиболее распространенные из них – ошибки, связанные с некорректным доступом к элементам конкретного представления объекта. Кроме того, увеличивается и теряет наглядность текст программы, в котором многократно повторяются аналогичные фрагменты явно запрограммированных действий над сложными данными (поиск, перебор, модификация и т.п.). По существу, такой стиль написания программ является одним из видов дублирования программных модулей, приносящим, возможно, больше вреда, чем полное дублирование программных разработок, так как при повторении сходных фрагментов велика вероятность внесения в программу ошибок.

Для преодоления этих недостатков стиля программирования при разработке больших программ используются технологические принципы модульного программирования и абстрактных типов данных. В работах по

этим направлениям сложился общий подход к декомпозиции программы и ее частей (процедур, пакетов и других модулей) на *интерфейс* – внешнюю часть, определяющую внешний эффект выполнения модуля, видимый его пользователем, и *реализацию* – фрагмент программы, осуществляющий этот внешний эффект. Термин "интерфейс модуля" понимается здесь в смысле, определенном в гл. 3 применительно к АТД. Под *интерфейсом объекта* будем понимать его имя и совокупность интерфейсов модулей, реализующих операции над этим объектом, т.е. определяющих видимое (абстрактное) поведение объекта.

В языках, имеющих средства модульного программирования и АТД (Модула-2, Клу, Ада и т.п.), принцип разделения на интерфейс и реализацию поддерживается языковыми конструкциями (модуль, пакет, кластер). Однако для практических приложений при разработке больших программ этого недостаточно. Проектирование и разработка интерфейса – весьма сложная часть процесса программирования, требующая использования четкой технологической дисциплины: регламентации структуры интерфейса, состава и назначения его программных модулей, горизонтального и вертикального расслоения и т.п. Эти регламентации могут зависеть от предметной области и класса решаемых задач. Иными словами, если от программиста требуется применение принципов модульного программирования в реальных задачах, он должен не только знать эти общие принципы и средства их выражения в инструментальном языке, но и иметь концептуальные средства для описания структуры технологического пакета (модуля) и его интерфейса на содержательном уровне.

Некоторые шаги в этом направлении предприняты в языке Клу (см. гл.3): программисту рекомендуется включать в состав интерфейса кластера операции *create* (генератор объектов), *equal* (сравнение объектов на совпадение), *similar* (сравнение объектов на сходство поведения) и *copy* (копирование объектов). Кроме того, в языке Клу введена возможность доопределения операций со стандартными обозначениями (+, * и т.п.) в их естественном смысле для определяемых типов данных. Благодаря этим простым концептуальным средствам программист имеет, например, готовый рецепт для определения кластера "матрица": он должен содержать операции *create*, *equal*, *similar* и *copy*; арифметические операции *add* (+), *mul* (*) (интерфейс всех перечисленных операций известен), а также может иметь другие специфические операции.

Сходные принципы используются и в более раннем языке *EL*/1 [86]: для каждого определяемого типа требуется указать набор стандартных процедур над объектами этого типа (генерация, присваивание, преобразование, вывод), которые затем в программе обозначаются традиционными знаками операций или используются неявно.

Следует учитывать, однако, особенности практики программирования, описанные в п. 3.1: недостаточную распространенность языков с АТД, инерцию программистов и их тяготение к более привычным языкам и, наконец, тот факт, что программист (как правило, связанный с определенным классом ЭВМ) будет применять для программирования только те языки, которые на ней реализованы, даже если это только Бейсик или Фокал. Поэтому концептуальные средства для проектирования и разработки интерфейса объектов не должны быть связаны только с каким-либо

конкретным языком программирования. Специалист-технолог должен определить для каждого языка и системы программирования подмножество конструкций языка и директив управления системой, на котором будут выражаться концепции объекта и его интерфейса, либо определить, что данный язык или система адекватных средств не имеет. Например, в языке Модула-2 есть подходящая конструкция "модуль", а в языке Паскаль описание интерфейса технологического пакета приходится моделировать другими средствами. Такого рода знания должны быть либо введены в технологический программный комплекс, поддерживающий данную технологию, либо изложены в форме методического руководства.

5.3.2. Цели ТИП-технологии. Предлагаемые технологические принципы программирования (ТИП-технология) являются развитием современных направлений методологии программирования: объектно-ориентированного подхода, модульного программирования и абстрактных типов данных. Целями ТИП-технологии являются:

– исключение дублирования в программных разработках не только на уровне больших комплексов программ, но и на уровне отдельных модулей; обеспечение повторного использования модулей;

– создание концептуальных средств и конкретных технологических пакетов для разработки программ в исследуемых предметных областях;

– поддержка жизненного цикла больших коллективных программных разработок на основе простых концептуальных средств выражения интерфейса между частями программ;

– повышение надежности программ; исключение ошибок, связанных с некорректным обращением к данным;

– улучшение наглядности и познаваемости программ;

– облегчение сопровождения и модификации программ.

ТИП-технология в ее настоящем виде, описываемом в данной главе, еще не вполне оформилась как технология программирования в полном современном понимании (п. 5.1) и для нее не накоплено столь значительного опыта использования, как, например, для Р-технологии. Однако уже имеющийся опыт применения ТИП-технологии (п. 5.5) – в частности, достигнутое резкое повышение производительности труда программистов, улучшение наглядности и надежности программ, использование готовых ТИПов в других программах – свидетельствует о ее перспективности для разработки трансляторов и, возможно, во многих других задачах, связанных с обработкой сложных структур данных.

5.3.3. Технологический инструментальный пакет (ТИП). Обзор ТИП-технологии. Основным концептуальным средством рассматриваемой технологии программирования является ТИП – *технологический инструментальный пакет.* ТИП определяет *интерфейс объекта* (совокупность *операций* над ним) и содержит в инкапсулированном виде его *конкретное представление* и *реализацию* операций. Новизна этого понятия, по сравнению с обычным понятием пакета (**package**), состоит в том, что горизонтальное и вертикальное слоение интерфейса, его форма и использование регламентированы. Интерфейс ТИПа должен содержать следующие *вертикальные слои* (группы модулей), соответствующие описанным выше основ-

ным классам операций над объектами:

– *интерфейс доступа*;

– *интерфейс генерации*;

– *интерфейс модификации*;

– *интерфейс вывода.*

Интерфейс также должен иметь следующие *горизонтальные слои* (уровни абстракции):

– *уровень представления* (*уровень* 0) – уровень работы в терминах элементов конкретного представления объекта;

– *уровень определения* (*уровень* 1) – уровень работы в терминах атрибутов объекта, указанных в его определении;

– *концептуальный уровень* (*уровень* 2) – уровень работы в терминах абстрактной концепции объекта или его реального прообраза, моделируемого данным объектом; например, для объекта "множество" – в терминах его элементов и теоретико-множественных операций, "дерево" – узлов, "самолет" – габаритов и характеристик полета, "вычислительная система" – виртуальных ресурсов.

Для обеспечения инкапсуляции и защиты от некорректного доступа к данным в ТИП-технологии введена следующая технологическая дисциплина использования модулей интерфейса в соответствии с их уровнями абстракции:

– внутри ТИПа модуль уровня n (n = 1,2) может использовать только модули уровней n и n – 1;

– при использовании (вызове) модуля из того же или другого ТИПа информационная связь с ним осуществляется только через аргументы, результаты и (для языков, имеющих средства обработки ситуаций) ситуации;

– из других ТИПов доступны только модули уровней 1 и 2.

Модули интерфейса ТИПа должны иметь небольшой объем (порядка 1–10 элементарных конструкций инструментального языка – описаний, операторов или вызовов других модулей). Включение того или иного модуля в ТИП определяется соображениями удобства и частоты возможного использования (а не "минимальности" набора модулей интерфейса).

Программа, разрабатываемая по ТИП-технологии, представляет собой совокупность ТИПов с межмодульными связями, соответствующими регламентациям.

П р о е к т и р о в а н и е и р а з р а б о т к а п р о г р а м м по ТИП-технологии – сложный циклический процесс, в котором чередуются этапы декомпозиции задачи на ТИПы, проектирования и реализации отдельных ТИПов и интерфейсов между ними. Начальный этап работы состоит в анализе накопленной библиотеки ТИПов для данной предметной области и выделении в решаемой задаче типовых частей, которые могут быть решены средствами уже имеющихся ТИПов либо их простых модификаций. На этом же этапе выделяются другие части задачи, которые, по экспертной оценке руководителя проекта или системного аналитика, выполняющего начальное проектирование, могут быть решены известными методами путем конструирования новых ТИПов. При этом, как правило, не удается заранее предусмотреть набор всех ТИПов, необходимых для решения данной задачи. Система развивается как открытая модульная среда путем кон-

струирования или модификации ТИПов, в процессе которого проектируются и реализуются интерфейсы между ними. Начальное проектирование системы и проектирование интерфейсов – наиболее сложные и ответственные этапы ТИП-технологии. Первый из них носит неформальный характер; он может быть успешно выполнен только специалистом, хорошо знающим задачу в целом. Для систематизации выполнения второго этапа ТИП-технология предоставляет концептуальные средства – регламентации структуры и слоения интерфейса. Подчеркнем, что эти средства облегчают не только первоначальную разработку ТИПа, но и его последующие модификации, неизбежные в любом программном проекте.

Разработка ТИПа выполняется снизу вверх, начиная с конкретного представления и интерфейса уровня представления. Такой стиль программирования позволяет, после проектирования и реализации уровня определения, абстрагироваться от деталей конкретного представления и постепенно надстраивать интерфейс до наиболее удобного уровня обращения к модулям. Кроме того, обеспечивается наиболее высокое качество и эффективность программ, так как программирование снизу вверх позволяет учесть особенности конкретного представления и эффективно реализовать модули интерфейса.

Реализация ТИПов выполняется средствами инструментального языка программирования, без расширения языка какой-либо специальной конструкцией. Рекомендации по выбору языковых средств для реализации ТИПов и применения ТИП-технологии для конкретного языка программирования должны быть даны в методической документации, составленной специалистом-технологом, или непосредственно в диалоговом режиме технологическим комплексом.

Программная поддержка ТИП-технологии осуществляется технологическим комплексом ТИП (п. 5.6). Информация об интерфейсе ТИПа и реализации отдельных модулей представляется на экране дисплея в символьном многооконном виде. Контролируется соблюдение технологической дисциплины (уровней, информационных связей и т.д.). Для облегчения работы программиста используется система меню, отображающих структуру интерфейса в виде фреймов, и элементы автоматического синтеза наиболее простых модулей.

Подробное описание ТИП-технологии дано в пп. 5.4–5.6. Примеры разработки программ по ТИП-технологии приведены в 5.5.

5.3.4. Сравнение с другими технологиями. ТИП-технология отличается от других рассмотренных технологий (п. 5.2) более тщательной разработкой понятия интерфейса, которое составляет одну из основ модульного программирования. Концепции АТД и пакета, развитием которых является ТИП-технология, сами по себе не обеспечивают семантической декомпозиции задачи на модули. Они дают только модульный "каркас" программы и нуждаются в концептуальных средствах, которые на содержательном уровне дают основу для разбиения задачи на модули. ТИП-технология (хотя она, как и любая технология, не может обеспечить разбиение на модули для любой задачи), предоставляет концептуальные средства для решения задач, связанных с обработкой сложных структурных данных. Набор этих средств может пополниться по мере накопления опыта программирования в исследуемой области.

Концепция вертикального слоения также учитывается в ТИП-технологии. С позиций ТИП-технологии добавление нового вертикального слоя в программную систему можно рассматривать как разработку нового ТИПа (или ТИПов), в интерфейс которого входят модули, реализующие компоненты слоя. Сами эти компоненты представляются в наглядной форме — вызовами модулей из интерфейса ТИПа. Вертикальное слоение введено и внутри ТИПа (интерфейс доступа, генерации, модификации и вывода).

ТИП как технологическая единица принципиально отличается (как подходом к программированию, так и назначением) от Р-схем — основных технологических единиц Р-технологии. В Р-схеме основное внимание уделено логической и управляющей структуре модуля, т.е. его реализации. ТИП-технология основное внимание уделяет интерфейсу; метод реализации конкретных модулей ею не регламентируется, поэтому реализация модуля может быть выполнена любым привычным для программиста способом — в частности, с помощью Р-схем. По-видимому, для описания интерфейса символьная мнемоническая форма (имена модулей, их аргументов, результатов и ситуаций) более удобна, чем графическая, так как логическая структура интерфейса проста и не требует графического изображения.

5.4. Методы разработки программ по ТИП-технологии

5.4.1. Требования к языковым средствам. Для применения ТИП-технологии программист может использовать привычный для него язык. Рассмотрим необходимые для этого возможности инструментального языка. Общий подход к ним заключается в следующем: применение ТИП-технологии не должно ухудшать эффективность программы.

Для *конкретного представления* объекта необходимо наличие в языке *описаний простых переменных и массивов*, представляющих атрибуты объекта. Например, для представления таблицы необходимо как минимум описание массива и простой переменной (текущего указателя). Наличие в языке описаний структур (записей) не обязательно, так как их можно смоделировать через переменные и массивы. Более того, использование записей может ухудшить эффективность программы, если оно не оптимизируется компилятором: например, если для представления таблицы использовано не два описания, как указано выше, а одно описание переменной-записи, содержащей поле-массив и поле-простое значение, то для доступа к компонентам представления могут потребоваться лишние команды. Поэтому рекомендуется распределенное описание конкретного представления, а инструментальный язык должен быть достаточно гибким, чтобы обеспечить эту возможность. В языке Клу, например, конкретное представление кластера (rep) должно быть задано одним обозначением типа, т.е. одним объектом (для сравнения, в языке Альфард распределенное описание конкретного представления допускается). Для сокращения числа команд доступа к объекту, представляемому записью, полезен механизм динамической установки локального контекста имен полей записи, который в языке Паскаль осуществляется оператором with.

Другой важный вопрос, связанный с конкретным представлением, — это его инкапсуляция (скрытие от пользователя). Инкапсуляцию обеспе-

чивает лишь сравнительно небольшое число языков программирования. Поэтому принцип инкапсуляции становится необходимым элементом технологической дисциплины, контролируемым лишь технологическим комплексом ТИП.

Для *описания модулей* интерфейса в инструментальном языке необходимо наличие простейших средств модульного программирования – процедур и текстовых макросов (открытых процедур) с параметрами. Одних процедур недостаточно, так как при малом размере модуля, рекомендуемом ТИП-технологией, накладные расходы на обращение к нему как к процедуре могут быть слишком велики. Вместо макросов возможно использование открытых процедур: например, текстов языка Эль-76, процедур *inline* языка Ада, квазипроцедур языка Ярмо [22]. В системах программирования Паскаль-Эльбрус (гл. 2) и Клу-Эльбрус (гл. 3) для повышения эффективности программ при применении ТИП-технологии введен специальный режим открытой подстановки процедур. Выбор способа реализации конкретного модуля (процедура или открытая процедура) должен определяться программистом.

Инструментальная система программирования также должна содержать специальные средства для *объединения ТИПов в одну программу*. Для этого предпочтительно использование таких средств, при которых программа, полученная в результате объединения, не содержит "лишних" динамических действий для связи ТИПов между собой (динамической сборки модулей и т.п.). Этому требованию удовлетворяет использование библиотечных вставок (операторов *include* в ПЛ/1, библиотечных карт в системах Эль-76 и Паскаль-Эльбрус) либо статической сборки (комплексации), как в системах Эль-76, Клу-Эльбрус и Модула-2-Эльбрус.

Таким образом, для применения ТИП-технологии инструментальная система программирования должна содержать:

– средства описания простых переменных и массивов во входном языке;

– средства описания процедур во входном языке и возможность (режим) организации открытых процедур;

– библиотечные вставки в текст программы на входном языке или средства статической сборки.

Большинство систем программирования, описываемых в данной работе, отвечает этим требованиям. Им не удовлетворяют языки ассемблерного уровня. В примерах разработки программ по ТИП-технологии используется язык Эль-76. Особенности применения ТИП-технологии при программировании на языках Паскаль-Эльбрус, Клу-Эльбрус и Модула-2-Эльбрус рассмотрены в пп. 5.4.7–5.4.9.

5.4.2. Этапы разработки программ. При традиционном подходе, характерном для программирования сверху вниз с использованием АТД, пошаговая разработка программы выполняется следующим образом:

1) разрабатываются модули более высокого уровня абстракции, использующие новый АТД;

2) проектируется новый АТД и его интерфейс (например, АТД "множество" с операциями "объединение", "пересечение" и "принадлежность");

3) разрабатывается конкретное представление и реализация операций АТД.

Такая последовательность этапов разработки приводит к тому, что при проектировании интерфейса не учитываются ни характер его использования (с точки зрения практического программирования), ни особенности конкретного представления (т.е. прежде всего эффективность доступа). При этом проектировщик интерфейса исходит не из практических соображений удобства использования интерфейса и эффективности его реализации, а из его математического или прикладного смысла. Таким образом, неправильное использование АТД может привести к разработке программ, пригодных лишь в качестве демонстрационных примеров, но неэффективных и неприменимых в реальных задачах: например, АТД "множество" с реализацией в виде массива, в котором операция "принадлежность" реализуется линейным поиском по всему множеству, а операция "объединение" – копированием аргументов в новый массив. Опыт разработки больших программ (например, [65, 68]) показывает, что при программировании сверху вниз невозможно заранее учесть все детали, прежде всего – определить интерфейс, и следовательно, неизбежны модификации программы, выполняемые снизу вверх. Кроме того, при программировании сверху вниз менее удобно выполнять отладку модулей, так как приходится моделировать их глобальную среду, что, как правило, неадекватно отражает реальную обстановку, в которой эти модули будут работать. Программирование снизу вверх более удобно и естественно, так как обеспечивает более высокое качество реализации, учет всех деталей и отражает реальный процесс накопления знаний в некоторой области (от частных к более общим) в виде совокупности модулей. С другой стороны, в любом программном проекте должен присутствовать первоначальный этап выделения частей, для которых уже имеются готовые модули или методы. Без такого этапа невозможно начать работу коллектива программистов. Он должен выполняться наиболее квалифицированным специалистом, знающим задачу в целом. В ТИП-технологии эти части задачи выражаются в виде ТИПов.

Программа в ТИП-технологии проектируется и разрабатывается в виде совокупности ТИПов, каждый из которых воплощает знания и опыт программиста-разработчика данного ТИПа в решении аналогичных подзадач (например, организация таблиц в трансляторах, организации базы знаний в экспертных системах). По мере накопления опыта разработки в какой-либо области создаются библиотеки ТИПов, которые могут быть использованы в последующих разработках в этой области.

Процесс проектирования и разработки программ в ТИП-технологии можно условно разделить на ряд этапов, которые в реальных разработках могут выполняться циклически и могут быть четко не отделены друг от друга.

1. Проектирование общей структуры системы:

– распознавание "известных" подзадач и поиск в библиотеке уже разработанных ТИПов, решающих эти подзадачи и пригодных для использования в данной системе;

– частичная первоначальная декомпозиция задачи, решаемой системой, на подзадачи, методы решения которых наиболее ясны, и определение необходимой совокупности ТИПов, решающих этих подзадачи;

– проектирование взаимосвязей между ТИПами (проектирование интерфейса основных связующих модулей).

Данный этап выполняется руководителем проекта и ведущими системными аналитиками, носит неформальный характер и лишь в незначительной степени может быть автоматизирован технологическим комплексом ТИП: руководитель имеет возможность просмотра библиотеки ТИПов; проектирование ведется при помощи простой системы меню, а результат его запоминается в базе данных проекта.

Например, при проектировании транслятора обычно выделяются ТИПы для реализации следующих функций: ввода-вывода; лексического анализа; синтаксического и семантического анализа отдельных групп конструкций входного языка (описаний, операторов, выражений, переменных, обозначений типов и т.п.); работы с таблицами (для каждой таблицы определяется свой ТИП); генерации кода (в качестве ТИПов для генерации кода в системе "Эльбрус" используются системные технологические пакеты ИНТЕРФОК и ИНТЕРКОД); диагностики ошибок и других сервисных функций.

2. Проектирование и разработка ТИПов, составляющих систему. Данный этап может выполяться параллельно несколькими программистами, имеющими наивысшую квалификацию в решении соответствующих подзадач. При выполнении этого этапа уточняются интерфейс каждого ТИПа и его взаимосвязи с другими ТИПами. Разработка каждого ТИПа выполняется как последовательность этапов:

2.1) разработка конкретного представления;

2.2) проектирование и реализация интерфейса уровня представления;

2.3) проектирование и реализация интерфейса уровня определения;

2.4) проектирование и реализация интерфейса концептуального уровня.

Выполнение этапа 2.4) может быть распараллелено с разработкой других ТИПов, так как после выполнения этапов 2.1)–2.3) основной интерфейс ТИПа уже зафиксирован. Интерфейс, спроектированный и реализованный по ТИП-технологии, учитывает свойства объекта, его представление, частоту и характер его возможного использования. Опыт показывает [65], что такая последовательность этапов наиболее удобна для разработки адекватного и эффективного реализуемого интерфейса.

Проектирование и реализация интерфейса уровня определения (при известном конкретном представлении) носит в значительной степени рутинный характер и автоматизируется технологическим комплексом ТИП. Наприме, если ТИП предназначен для управления некоторым объектом табличного характера и известно его конкретное представление (совокупность классов элементов таблицы и полей, составляющих элемент каждого класса), то модули реализации интерфейса доступа, генерации, модификации и вывода уровня определения синтезируются автоматически комплексом ТИП. Это почти полностью избавляет программиста от технической работы.

Разработка интерфейса концептуального уровня наиболее сложна, так как требует анализа возможных часто выполняемых действий над объектом (поиска отдельных атрибутов, их перебора и т.п.). Она не может быть автоматизирована и выполняется "вручную" с использованием интерфейса уровня определения.

Отметим, что ТИП-технология предлагает не простую перестановку этапов разработки (по сравнению с методом "сверху вниз"), а принципиальное изменение точки на интерфейс и реализацию. При использовании метода "сверху вниз" в больших коллективах программистов имеется тенденция разделения на "спецификаторов" (наиболее квалифицированных специалистов, только проектирующих интерфейсы и самостоятельно не программирующих) и "реализаторов" (разработчиков конкретного представления и реализации, т.е. собственно программистов; для этой работы часто используются менее квалифицированные кадры). Такое ошибочное отношение к интерфейсу как к более важной компоненте и к реализации как к второстепенной может привести к ухудшению качества программ и несоответствию (как по качеству, так и по выполняемым функциям) реализованной версии системы ее первоначальному проекту или техническому заданию.

В ТИП-технологии подход к разработке совершенно иной. Интерфейс и реализация – тесно связанные, одинаково важные компоненты программы. Наибольшее внимание придается реализации и интерфейсу нижних уровней ТИПа, которые разрабатываются специалистом, имеющим наибольший опыт решения аналогичных подзадач и, следовательно, будут иметь наивысшее качество, что в конечном счете обеспечит и высокое качество всей программы.

5.4.3. Конкретное представление и реализация разрабатываются на инструментальном языке проекта, удовлетворяющем условиям, сформулированным в п. 5.4.1. Разработка конкретного представления – начальный этап разработки ТИПа. Конкретное представление должно обладать следующими основными свойствами:

– адекватно отображать реальные свойства объекта;

– обеспечить возможность эффективного доступа к атрибутам.

С целью повышения эффективности доступа сосредоточенное представление (в виде описания одной сложной структуры данных с несколькими компонентами) рекомендуется заменить распределенным (в виде нескольких отдельных описаний компонент). Например, стек рекомендуется представлять описаниями массива и переменной-текущего указателя, а не одним описанием записи из двух полей указанных типов. Распределенное описание конкретного представления в сочетании с реализацией интерфейса уровня определения в виде открытых процедур обеспечивает ту же степень наглядности, но бóльшую эффективность.

П р и м е р. ТИП для обработки анкет на языке Эль-76. Этот пример будет использоваться в качестве модельного в течение всей остальной части главы. Необходимо спроектировать и реализовать совокупность операций обработки анкетной информации. Число анкет может быть произвольным. Анкета должна включать следующие данные: фамилию, инициалы, дату рождения, информацию о супруге и пол.

Кроме того, для мужчин указывается воинское звание, для женщин – число детей.

·На языке Эль-76 информацию табличного характера наиболее удобно представлять в виде вектора переменной длины (п. 1.6.2). Элемент таблицы (анкета) занимает целое число слов, что обеспечивает более эффективный доступ к атрибутам (индексация и выделение поля), чем при использовании упакованных переменных.

Представление элемента таблицы (анкеты) имеет переменную длину, зависящую от длины фамилии. Язык Эль-76 позволяет наиболее экономно распределить память "вручную" на уровне представления с помощью описаний полей. Более традиционные языки (например, Паскаль) не имеют средств распределения памяти уровня представления; оно выполняется системой программирования, а программисту предоставляется возможность описать структуру элемента таблицы с помощью типа "запись" без какой-либо информации о желаемом размещении полей (даже указанный в описании порядок полей может не соблюдаться компилятором).

Для распределения памяти внутри элемента определяется максимальная длина значения каждого атрибута в упакованном виде: инициалы – по 8 битов, год – 11 битов, месяц – 4 бита, число рождения – 5 битов, признак "женат" – 1 бит, ссылка на информацию о супруге – 20 битов (индекс в векторе), пол – 1 бит, длина фамилии – 6 битов. Таким образом, постоянная часть таблицы занимает 64 бита, т.е. одно слово. Сменная часть элемента таблицы: для мужчин – признак "военнообязанный" (1 бит) и звание (числовой код – 8 битов); для женщин – число детей (5 битов). Фамилию целесообразно размещать после всех полей постоянной длины.

Приведем полное описание конкретного представления ТИПа "анкета":

```
конст % --- длины таблицы анкет --- %
    дланк     = 512,        % начальная длина таблицы анкет
    максанк   = 5120;       % максимальная длина таблицы анкет
поле % --- поля таблицы анкет --- %
    % ++++++++++++++++++ слово 0: постоянная часть +++++++++++ %
    аимяп     = [63 : 8],   % инициал имени
    аотчп     = [55 : 8],   % инициал отчества
    агодп     = [47 : 11],  % год рождения
    амесяцп   = [36 : 4],   % месяц рождения
    ачислоп   = [32 : 5],   % число рождения
    аженатп   = [27],       % признак "женат"
    асупругп  = [26 : 20],  % ссылка на анкету супруга
    аполп     = [6],        % пол
    адлфамп   = [5:6],      % длина фамилии
  % ++++++++++++++++++ слово 1: мужчины ++++++++++++++++++ %
    авоенп    = [63],       % признак "военнообязанный"
    азванuеп  = [62 : 8],   % военское звание
 % +++++++++++++++++ слово 1: женщины (пол=1) ++++++++++++ %
    ачисдетейп= [63 : 5];   % число детей
```

```
  % фамилия хранится со второго байта слова 1 (с бита 47) и
  % занимает длфам байтов
ф64 % ——— тело таблицы анкет ——— %
    анк: = лок вект [дланк:максанк] ф64, % таблица анкет
    уканк: = 1,                          % текущий указатель
    теканк;                              % текущая анкета
```

Таблица анкет представлена телом таблицы – вектором переменной длины – и двумя указателями – на текущее свободное слово и на текущую обрабатываемую анкету. Анкета характеризуется индексом *а* ее начального слова. При описании полей используется следующая мнемоника: первая буква (*а*) – от названия ТИПа (*анкета*), последняя (*п*) – от названия элемента (поле), стандартная для всех описаний полей.

Этап создания конкретного представления автоматизируется технологическим комплексом ТИП: описания полей синтезируются по заданным именам и размерам (типам) полей.

5.4.4. Проектирование и разработка интерфейса. Проектирование интерфейса ТИПа – второй этап разработки ТИПа. Интерфейс проектируется как совокупность четырех основных групп модулей.

– интерфейс доступа;

– интерфейс генерации;

– интерфейс модификации;

– интерфейс вывода.

Возможны и другие, дополнительные виды интерфейса, отражающие специфику объекта: например, интерфейс преобразования данных из одной формы в другую в СУБД; интерфейс по управлению в системах параллельной обработки данных. Классификация интерфейса зависит от предметной области.

Проектирование и реализация начинаются с интерфейса доступа: разрабатывается его уровень представления. Это позволяет своевременно обнаружить недостатки конкретного представления (неэффективность доступа, недостаточность информации) и модифицировать его совместно с интерфейсом доступа. Далее проектируется и реализуется интерфейс генерации и интерфейс модификации; как показывает опыт использования ТИП-технологии [65], он имеет наиболее сложную структуру, и необходимость его изменения также может повлиять на конкретное представление. Интерфейс и реализацию средств вывода наиболее удобно разрабатывать после разработки уровней представления и определения интерфейса доступа; это позволяет производить отладочные печати в ходе любой дальнейшей доработки.

Проектирование и разработка интерфейса выполняется снизу вверх: уровень представления – уровень определения – концептуальный уровень. При такой последовательности разработки программист (после проектирования и реализации уровня определения) абстрагируется от деталей конкретного представления и в дальнейшем использует для доступа, генерации и модификации только модули интерфейса уровня определения. На основе реализованных нижних уровней интерфейса значительно облегчается дальнейшее проектирование и разработка ТИПа, повышается надежность, улучшается наглядность и уменьшается объем программы.

С п е ц и ф и к а ц и я ТИПа и е г о м о д у л е й. В процессе проектирования фиксируется назначение ТИПа и форма обращения к его интерфейсу. Тексту программных модулей ТИПа должна предшествовать подробная спецификация в виде комментария, которая должна содержать:

– имя и назначение ТИПа, его версию, принадлежность к программной системе, условия использования, информацию об авторе;

– спецификации модулей интерфейса ТИПа: имя модуля, имена аргументов и результата; информацию об их типах; принадлежность модуля к определенному виду интерфейса; его уровень; краткую словесную спецификацию эффекта выполнения модуля и возможных исключительных ситуаций (если она не ясна из предшествующей информации).

Имя модуля должно отражать его назначение и принадлежность к определенному ТИПу (например, в виде префикса), если это не обеспечивается автоматически средствами языка. Например, на языке Клу обращение к операции выборки атрибута "*фамилия*" из анкеты *а* может иметь вид

анкета $ фамилия (а),

где *анкета* – имя кластера (ТИПа) для обработки анкет. На языке Эль-76 модуль, реализующий аналогичную операцию, может называться *афам*, где *а* – префикс, обозначающий принадлежность к ТИПу анкета (язык Эль-76 не допускает пробелов и подчеркиваний внутри идентификатора).

В ТИП-технологии допускается два вида параметров модуля: *параметры-значения* (используемые внутри модуля) и *параметры-переменные* (модифицируемые внутри модуля). В реализации модуля не должны использоваться имена глобальных величин, кроме:

– имен модулей текущего и предыдущего (более низкого) уровня данного ТИПа;

– имена модулей уровня определения и концептуального уровня других ТИПов;

– имен параметров данного модуля;

– имен компонент конкретного представления (только в реализации модулей интерфейса уровня определения).

Таким образом, любой побочный эффект, связанный с модификацией глобальных переменных в некотором модуле, должен быть явно выражен через параметры-переменные. Имена параметров-переменных в спецификации модуля помечаются символами @: @ *смещ*.

П р и м е р спецификации модуля из ТИПа анкета:

% *амуж* (*фам, и, о, год, мес, число, женат, супруга, воен, зв*) → *а*
% формирует анкету лица мужского пола и записывает ее в табли-
% цу анкет. Вид интерфейса: генерация. Уровень: определения.
% Параметры: анкетные данные. Результат: ссылка на новую анкету

Интерфейс доступа состоит фактически не из трех, а из двух уровней (уровень представления обеспечивается средствами языка):

– доступ уровня представления (уровня 0) – непосредственное обращение к компонентам конкретного представления в терминах языка программирования. Использование доступа уровня представления разрешено ТИП-технологией только в реализации модулей уровня определения. Примеры доступа уровня 0: *уканк*+1, *анк* [*а* + 1]. [63 : 5], *анк*[*а*]. *аполп*.

Явное использование доступа этого уровня в других частях программы приводит к большому числу ошибок, поэтому ТИП-технологией оно запрещено;

– доступ уровня определения (уровня 1) – обращение к атрибутам без использования информации об их размещении. Модули доступа этого уровня наиболее критичны по эффективности, поэтому для их реализации необходимо использование текстовых макросов или открытых процедур. Например, реализация модуля доступа к атрибуту *год* ТИПа *анкета* на языке Эль-76 записывается в виде определения текста:

```
текст агод (а) = анк [а] . агодп
Использование:
если агод (х) >= 1965 то печстр (стр 8 "молодой") все
```

Реализация модулей доступа уровня определения, как наиболее простых, может синтезироваться технологическим комплексом ТИП по информации о конкретном представлении. Автоматический синтез модулей доступа уровня определения выполняется по желанию и указанию программиста в некоторых часто встречающихся случаях определения ТИПов (например, при обработке таблиц, списков и других структур данных);

– доступ концептуального уровня (уровня 2) – запросы к информации об объекте в форме, наиболее удобной пользователю ТИПа. Реализация модуля доступа этого уровня может содержать ассоциативный поиск. На данном уровне доступа от пользователя скрыто не только представление атрибутов и компонент объекта, но и метод их перебора (анализа). Примером модуля доступа концептуального уровня в языках с АТД является итератор (см. гл. 3), выполняющий перебор компонент сложного объекта способом, скрытым от пользователя.

Примеры модулей доступа концептуального уровня для ТИПа *анкета*:

```
текст
апенс (а, текгод) = % –– а – пенсионного возраста на текущий год? –– %
   агод (а) < = если апол (а) % женщина
                то текгод – 55
                иначе текгод – 60
                все, % ––– апенс ––– %
 афио (фам, имя, отч, нач) =
 % ––– ссылка на анкету заданного лица
 % (если такого нет, то                    0) ––– %
 % (поиск начинается с анкеты нач)           %
   до найдена, конецанкет
   цикл
   ф64 тек: = нач; % ссылка на текущую анкету
   если тек = 0 то конецанкет! (0)
   инес афам (тек) = /фам и аимя (тек) = имя
        и аотч (тек) = отч
   то найдена! (тек)
   иначе тек:=аслед (тек) % переход к следующей анкете
   все
   повторить % ––– афио ––– %
```

В примере предполагается, что модули доступа уровня определения *агод* (*а*), *апол* (*а*), *афам* (*а*) и *аслед* (*а*) уже спроектированы.

Рекомендуется програмировать все подобного рода запросы к объекту в виде модулей доступа концептуального уровня. В ходе разработки данный уровень пополняется, так как необходимость таких модулей трудно предусмотреть заранее.

Интерфейс генерации рассмотрим также снизу вверх, в соответствии с уровнями абстракции.

Интерфейс генерации уровня представления – это совокупность модулей управления памятью для формирования представления объекта или его компонент. Если конкретное представление состоит только из простых переменных, то уровень представления в интерфейсе генерации отсутствует (генерация осуществляется при исполнении описаний, составляющих конкретное представление). Если же при представлении использованы динамические структуры данных (объект имеет динамическую природу), то уровень представления в интерфейсе формирования абстрагирует от особенностей распределения памяти и от возникающих при этом исключительных ситуаций.

Пример модуля интерфейса генерации уровня 0 для ТИПа *анкета*: отведение памяти для анкеты длиной *ск* слов.

```
текст адайанк (ск) =
%——— отведение памяти для анкеты длиной ск слов. ——— %
%      результат: ссылка на начальное слово анкеты.      %
 (если уканк+ск >= длина анк
  то   %——— превышена текущая длина тела таблицы анкет
       если   уканк+ск >= максанк
       то     %——— память исчерпана ——— %
              ошибка (стр8 "нет памяти для анкеты")
       иначе  %——— увеличить длину тела таблицы анкет
              анк: =измдлину (анк, дланк)
       все
  все;
  теканк: =уканк;
  уканк: = * + ск;
  теканк ) % ——— адайанк ——— %
```

Для типичных случаев (например, для обработки объектов табличного типа) технологический комплекс ТИП синтезирует модули интерфейса генерации уровня представления по указанию программиста, так как все они имеют аналогичную структуру.

Интерфейс генерации уровня определения состоит из модулей генерации компонент объекта по заданным значениям атрибутов. Для управления памятью при реализации интерфейса генерации уровня определения используются модули интерфейса генерации предыдущего уровня.

Пример модуля интерфейса генерации уровня определения для ТИПа *анкета* (спецификация этого модуля приведена выше):

```
текст
  амуж (фам, и, о, год, мес, число, женат, супруга, воен, звание) =
     начало % ——— отведение памяти для анкеты ——— %
```

```
    теканк: = адайанк) (длина фам + 1)/:8 + 2);
  % --- заполнение анкеты --- %
    анк [теканк] :=
        (аполп:    0,           % пол – мужской
         аимяп:    и,           % инициал имени
         аотчп:    о,           % инициал отчества
         агодп:    год,         % год рождения
         амесяцп: мес,          % месяц рождения
         ачислоп: число,        % число рождения
         аженатп: женат,        % признак "женат"
         асупругп: супруга,     % ссылка на анкету жены
         адлфамп: длина фам);   % длина фамилии
    анк [теканк: + 1] :=
        ·(авоенп:     воен,     % воинская обязанность
          азваниеп:   звание);  % воинское завание
    анк [ф8 (теканк+1)*8 + 2:] <:= фам; % фамилия
  % --- результат: --- %
    теканк
конец % --- амуж --- %
```

В наиболее простых случаях (если объект имеет статическую структуру или состоит из компонент с атрибутами простых типов) технологический комплекс ТИП по указанию программиста автоматически синтезирует модули интерфейса генерации уровня определения.

Интерфейс генерации концептуального уровня состои из модулей генерации объекта или его компонент в терминах его абстрактных свойств или реального прообраза: например, для дерева – в терминах узлов, для списка – элементов, для табличного представления описаний входного языка в компиляторе – в терминах абстрактного синтаксиса описаний. Примером модуля интерфейса генерации концептуального уровня для ТИПа *анкета* мог бы быть модуль, формирующий список анкет по вводимым анкетным данным.

Интерфейс модификации предназначен для изменения состояния объекта – его атрибутов, абстрактных компонент и связей между ними. Некоторые из атрибутов могут быть не модифицируемыми (константами) – например, атрибут *пол* ТИПа *анкета*, – что должно быть отражено в описании интерфейса модификации.

Уровень представления образуют средства инструментального языка для изменения элементов конкретного представления (различные формы присваивания). Они используются при реализации модулей следующего уровня.

Уровень определения образуют модули изменения значений атрибутов. Этот уровень интерфейса модификации наиболее удобно было бы использовать, сохраняя традиционную форму ":=" для обозначения операции модификации:

атрибут (*объект*) := *значение*

Однако при этом символ ":=" должен пониматься как знак операции уровня определения. В левой части этого присваивания наиболее удоб-

но было бы использовать модуль уровня определения, осуществляющий доступ к заданному атрибуту объекта. Однако это возможно лишь в случае, если вызов модуля доступа *атрибут* (*объект*) обозначает, в зависимости от позиции в программе, либо значение, либо адрес атрибута. Этим свойством обладает, например, текстовый макрос языка Эль-76, правая часть определения которого является обозначением переменной:

текст *аженат* (*а*) = *анк* [*а*] . *аженатп*

Поэтому для модификации атрибута *женат* можно использовать конструкцию

аженат (*а*) := 1

Другой вариант решения этой проблемы предлагает язык Клу (см. гл. 3), в котором символом ":=" могут обозначаться не только собственно присваивания (перестановка ссылок), но и операции *store* (изменение компоненты объекта, подобного массиву) и *set_f* (изменения значения поля *f* объекта, подобного записи). В языке Паскаль вызов функции не может использоваться в левой части присваивания, поэтому операции модификации уровня определения должны записываться в процедурной форме (присваивания элементам массивов и полям записей в языке Паскаль являются операциями уровня представления).

Атрибуты-константы, не подлежащие модификации (например, атрибут *пол* в анкете), должны быть соответствующим образом определены, если это позволяет используемый язык, либо специфицированы как константы с помощью комментариев. Например, в расширенном языке Эль-76 [52] имеются средства описания полей-констант в объектах структурных типов; аналогичные средства имеются в языке Ада.

Концептуальный уровень интерфейса модификации составляют модули изменения абстрактных компонент объекта или их абстрактных связей.

П р и м е р. Модуль внесения в список анкет информации о заключении брака (входит в интерфейс модификации концептуального уровня).

```
текст абрак (фм, им, ом, фж, иж, ож) =
%––– внесение информации о заключение брака ––– %
  (конст муж = афио (фм, им, ом, аначанк),    % анкета мужа
         жена = афио (фж, иж, ож, аначанк);   % анкета жены
    аженат (муж) := 1;   асупруг (муж) := жена;
    аженат (жена) := 1;  асупруг (жена) := муж
  ), % ––– абрак ––– %
```

В модуле *абрак* используются (в форме присваивания) операции интерфейса модификации уровня определения для атрибутов *женат* и *супруг*. Модуль интерфейса доступа уровня определения *аначанк* выдает ссылку на начало списка анкет.

Интерфейс вывода обеспечивает вывод информации об объекте в наиболее наглядной форме, удобной для отладки программы и получения результатов. Он также состоит из трех уровней: представления, определения и концептуального.

Уровень представления в интерфейсе вывода, как показал опыт применения ТИП-технологии [65], наиболее удобно оформлять в виде отдель-

ного ТИПа нижнего уровня, общего для всех ТИПов, который состоит из набора модулей для вывода простых объектов (чисел, логических значений, символов, строк), перехода на новую строчку при выводе и т.п. Например: *печцел* (*целое*), *печвещ* (*вещественное*), *печстр* (*строка*), *новстрок*. Этот ТИП абстрагирует от привязки к конкретному классу выводного устройства и позволяет выводить результаты на несколько устройств (например, на терминал и на АЦПУ). Он обеспечивает вывод элементов конкретного представления. Уровень представления в интерфейсе вывода может быть использован в других системах без каких-либо модификаций.

Уровень определения в интерфейсе вывода предназначен для вывода атрибутов объекта. Например, ТИП *анкета* может содержать модули *апечфам* (*фам*), *апечгод* (*год*) и т.д., реализация которых выражена через примитивы уровня представления. Модули уровня определения интерфейса вывода синтезируются технологическим комплексом ТИП по указанию программиста.

Концептуальный уровень интерфейса вывода обеспечивает вывод информации об объекте в целом или его крупных содержательных фрагментах в символьном виде, в терминах прообразов объектов. Например, для ТИПа *анкета* наиболее удобны следующие модули концептуального уровня интерфейса вывода: *апечанкеты* (*а*) – полный вывод анкеты; *апечанкет* (*от, до*) – вывод всех анкет из указанного диапазона.

Понятие вывода может трактоваться более широко. Например, ТИП *анкета* мог бы иметь модуль интерфейса вывода, записывающий список анкет в файл.

5.4.5. Пример. В заключение общего описания ТИП-технологии приведем полный текст ТИПа *анкета* для обработки анкет на языке Эль-76. Этот пример можно рассматривать и как завершение гл. 1 – пример разработки и оформления большого фрагмента программы на языке Эль-76.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                    Тип "анкета"                                    %
% обработка анкетной информации. Использует: ввод-вывод, сервис      %
% автор: В.О. Сафонов. Версия: 1.0. Дата: 30.08.87                   %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%-----------------конкретное представление ТИПа ------------%
    конст
      дланк = 512,          % начальная длина таблицы анкет
      максанк = 5120;       % ее максимальная длина
   ф64
      анк := лок вект [дланк: максанк] ф64, % таблица анкет
      уканк := 1,                  % текущий указатель таблицы
      теканк;                      % указатель текущей анкеты
%++++++++++++++++++++++++ поля анкеты: ++++++++++++++++++++++++%
   поле
% +++++++++++++++++ слово 0: постоянная часть +++++++++++++++++++%
     аимяп = [63 : 8],          % инициал имени
     аотчп  = [55 : 8],          % инициал отчества
     агодп  = [47 : 11],         % год рождения
```

```
    амесяц = [36 : 4],            % месяц рождения
    ачислоп = [32 : 5],           % число рождения
    аженатп = [27],               % признак "женат"
    асупругп = [26 : 20],         % ссылка на анкету супруга
    аполп = [6],                  % пол (0 – муж, 1 – жен)
    адлфамп = [5:],               % длина фамилии
%++++++++++++++++ слово 1: мужчины (пол = 0) ++++++++++++++ § %
    авоенп = [63],                % признак "военнообязанный"
    азваниеп = [62 : 8],          % воинское звание
%+++++++++++++++++ слово 1: женщины (пол = 1) ++++++++++++++++ %
    ачисдетейп = [63 : 5],        % число детей
%   символы фамилии хранятся, начиная с бита 47 слова 1 и зани-   %
%   мают длфам байтов

% ---------------- интерфейс ТИПА -----------------%
% ==================== интерфейс доступа ===================== %
% +++++++++++++++++ уровень определения (1) +++++++++++++++++ %

   текст % --- общая часть --- %
    аимя        (а) = анк [а]. аимяп,          % имя
    аотч        (а) = анк [а]. аотчп,          % отчество
    агод        (а) = анк [а]. агодп,          % год рождения
    амесяц      (а) = анк [а]. амесяцп,        % месяц рождения
    ачисло      (а) = анк [а]. ачислоп,        % число рождения
    аженат      (а) = анк [а]. аженатп,        % признак "женат"
    асупруг     (а) = анк [а]. асупругп,       % ссылка на супруга
    апол        (а) = анк [а]. аполп,          % пол (1-м, 0-ж)
    адлфам      (а) = анк [а]. адлфамп,        % длина фамилии
    адланк      (а) = а+2+(адлфам(а)+1)/:8,    % длина анкеты
    афам        (а) = % --- фамилия (дескриптор) --- %
          анк [ф8 (а + 2)*8 : адлфам (а)],
    аначанк = 1, % --- начало списка анкет --- %
% +++++++++++++++++++++++++ мужчины: +++++++++++++++++++++++ %
    авоен       (а) = анк [а+1]. авоенп,     % воинская обязанность
    азвание     (а) = анк [а+1]. азваниеп,   % воинское звание
% ++++++++++++++++++++++++ женщины: ++++++++++++++++++++++++%
    ачисдетей   (а) = анк [а+1]. ачисдетейп, % число детей
% +++++++++++++++++ уровень 2: концептуальный ++++++++++++++++ %
    апенс (а, текгод) = % --- а – пенсионер на текущий год --- %
        агод (а) < если апол (а) то текгод – 55
                   иначе текгод-60
                   все, % --- апенс --- %
    аслед (а) = % --- ссылка на следующую анкету или 0 ---%
        если аимя (а + адланк (а)) = 0 % данная анкета-последняя
        то 0
        иначе а + адланк (а)
        все, % --- аслед --- %
    афио (ф, и, о, нач) =
% ---поиск анкеты заданного лица, начиная с анкеты нач --- %
          до найдена, конецанкет
```

```
    цикл
        ф64 тек := нач;  % ссылка на текущую анкету
        если тек = 0 то конецанкет! (0)
        инес афам (тек) =/ф и аимя (тек) = и и аотч (тек) = отч
        то найдена! (тек)
        иначе тек := аслед (тек) % переход к следующей анкете
        все
    повторить, % --- афио --- %
% ==================== интерфейс генерации ==================== %
% +++++++++++++++++++ уровень представления (0) +++++++++++++++ %
адайанк (ск) = % --- отведение памяти для анкеты (ск слов)
   (если уканк+ск >= длина анк
    то  % --- превышена текущая длина таблицы анкет
         если уканк+ск >= максанк
         то    % --- память для анкет исчерпана --- %
               ошибка (стр8" нет памяти для анкеты")
         иначе % --- увеличение длины области памяти --- %
               анк := измдлину (анк, дланк)
         все
    все;
    теканк := уканк; уканк := * + ск;
    теканк), % --- адайанк --- %
% +++++++++++++++++++ уровень определения (1)+++++++++++++++++++%
    амуж (ф, и, о, год, мес, чис, жен, супр, воен, зв) =
    % --- создает анкету мужчины с заданными анкетными данными %
        (% --- отведение памяти --- %
          теканк := адайанк ((длина ф + 1)/:8 + 2);
         % --- заполнение анкеты --- %
          анк [теканк] :=
                 (аполп:     0,           % пол – мужской
                  аимяп:     и,           % имя
                  аотчп:     о,           % отчество
                  агодп:     год,         % год рождения
                  амесяцп:   мес,         % месяц рождения
                  ачислоп:   чис,         % число рождения
                  асупругп:  супр,        % ссылка на анкету жены
                  аженатп:   жен,         % признак "женат"
                  адлфамп:   длина ф);    % длина фамилии
          анк [теканк + 1] :=
                 . (авоенп:   воен,       % признак "военнообязанный"
                  азваниеп:  зв);         % воинское звание
          анк [ф8 (теканк + 1) * 8 + 2:] < := ф; % фамилия
          теканк % результат
        ), % --- амуж --- %
ажен (ф, и, о, год, мес, чис, зам, супр, чисдетей) =
%---создает анкету женщины (аналогично амуж) ---!
     (теканк := адайанк ((длина ф+1)/:8 + 2);
      анк [теканк] :=
          .(аполп: 1, аимяп: и, аотчп: о, агодп: год,
```

```
            амесяцп: мес, ачислоп: чис, асупругп: супр,
            аженатп: зам, адлфамп: длина ф);
       анк [теканк+1] := . (ачисдетейп: чисдетей); % число детей
       анк [ф8(теканк+1) * 8 + 2:] < := ф;
       теканк), % --- ажен --- %
% ---- интерфейс модификации ---- %
% ++++ уровень определения (1) ++++ %
% для атрибутов имя, отч, женат, супруг, воен, звание, чисдетей
% используется интерфейс в виде присваивания, например:
% аженат (а) := 1
% атрибуты год, месяц, число, пол – константы.
амодфам (а, ф) = % --- модификация атрибута фамилия --- %
      если анк [а]. адлфамп < длина ф
      то % -- усечение фамилии до размера анкеты
         анк [ф8 (а+1) *8 + 2:] <:= ф длиной анк [а]адлфамп
      иначе
         анк [а]. адлфамп := длина ф;
         анк [ф8 (а+1) *8 + 2:] <:= ф
     все, % --- амодфам --- %
% ++++++++++++++++ уровень 2: концептуальный ++++++++++++++++ %
абрак (фм, им, ом, фж, иж, ож) =
   %--- внесение информации о заключении брака --- %
        (конст муж = афио (фм, им, ом, аначанк),
               жена = афио (фж, иж, ож, аначанк);
         аженат (муж) := 1; асупруг (муж) := жена;
         аженат (жена) := 1; асупруг (жена) := муж
        ), % --- абрак --- %
% ==================== интерфейс вывода ==================== %
% +++++++++++++++++ уровень определения (1) ++++++++++++++++ %
апечимя (имя) = % --- вывод атрибута имя --- %
      (печсимв (имя); печсимв (".")),
апечотч (отч) = % --- вывод атрибута отчество --- %
      (печсимв (отч); печсимв (".")),
апечгод (год) = % --- вывод атрибута год --- %
      (печ8смв (" г.р.="); печцел64 (год)),
апечмесяц (месяц) = % --- вывод атрибута месяц --- %
      (печ8смв ("месяц:");
      выбор месяц из
         1: печ8смв ("январь"), . . .
         12: печ8смв ("декабрь")
      всевыб), % --- апечмесяц --- %
апеччисло (число) = % --- вывод атрибута число --- %
      (печ8смв ("число:"); печцел64 (число)),
апечженат (женат, пол) = % --- вывод атрибута женат --- %
      если пол то % --- женщина --- %
         если женат то печстр (стр8 "замужем")
         иначе печстр (стр8 "не замужем")
         все
```

```
        инес % --- мужчина --- %
             женат
        то   печ8смв ("женат")
        иначе печ8смв ("холост")
        все, % --- апечженат --- %
    апечсупруг (супруг, женат) = % вывод атрибута супруг --- %
        если женат то
             печ8смв ("супруг:<");
             печцел64 (супруг);
             печсимв (">")
        все, % --- апечсупруг --- %
    апечпол (пол) = % --- вывод атрибута пол --- %
        (печ8смв ("пол:");
        если пол то печ8смв ("жен") иначе печ8смв ("муж") все
      ), % --- апечпол --- %
    апечфам (фам) = % --- вывод атрибута фамилия --- %
        (печстр (фам)),
    апечвоен (воен) = % --- вывод атрибута "военнообязанный" -%
        (если не воен то печ8смв ("не") все;
         печстр (стр8 "военнообязанный")
         ), % --- апечвоен --- %
    апечзвание (зв, воен) = % --- вывод атрибута звание --- %
        если воен то
             выбор зв из
                1: печ8смв ("рядовой"),
                2: печ8смв ("сержант"), . . .
             всевыб
        все, % --- апечзвание --- %
    апеччисдетей (чисдетей) = % --- вывод атрибута "число детей"
        (печ8смв ("число"); печ8смв ("детей:");
         печцел64 (чисдетей)
         ); % --- апеччисдетей --- %
% ++++++++++++++++++ уровень 2: концептуальный ++++++++++++++++ %
    процедура апечанкеты = проц (а)
        % --- полная печать анкеты --- %
    (новстрок; печцел64 (а); печ8смв (") ");
    апечфам (афам (а)); апечимя (аимя (а)); апечотч (аотч (а));
    апечпол (апол (а)); апечгод (агод (а)); апечмесяц (амесяц (а));
    апеччисло (ачисло (а)); новстрок;
    апечженат (аженат (а), апол (а));
    апечсупруг (асупруг (а), аженат (а));
    если апол (а) то апеччисдетей (а)
    иначе апечвоен (авоен (а)); апечзвание (азвание (а), авоен (а))
    все;
    новстрок); % --- апечанкеты --- %
    процедура апечанкет = проц (от, до)
    начало
      новстрок;
      печстр (стр8 "====== список анкет ========= ");
```

```
    новстрок;
    до  конецанкет цикл
        ф64 тек:= от;
        если тек > до то конецанкет!
        иначе  апечанкеты (тек);
               тек: = аслед (тек),
        все
    повторить;
    новстрок;
    печстр (стр8 "====== конец списка анкет ======");
    новстрок
  конец; % --- апечанкет --- %
% --------------- конец ТИПа анкета ------------------- %
```

Для использования ТИПа *анкета* в программе на Эль-76 необходимо вставить в программу библиотечную карту:

¤*биб* //*анкета* ,

где //*анкета* – внешнее имя текстового файла, содержащего текст ТИПа *анкета*.

5.4.6. Использование языка Эль-76. Методы и примеры использования языка Эль-76 для программирования по ТИП-технологии описаны в пп. 5.4.3–5.4.5. Здесь лишь кратко суммируется эта информация и рассматриваются вопросы соотношения концепций ТИП-технологии и модульных объектов системы "Эльбрус".

Для программирования на языке Эль-76 по ТИП-технологии используются следующие конструкции:

– для конкретного представления – описания констант (длин векторов), переменных (простых атрибутов, счетчиков и указателей векторов) и полей (размещения атрибутов);

– для описания модулей интерфейса – текстовые макросы и процедуры. Текстовые макросы применяются для описания модулей небольшого объема, наиболее критичных по эффективности (например, интерфейса доступа уровня определения);

– для подключения ТИПов к использующей программе – библиотечные карты. Текст описания ТИПа рекомендуется хранить в отдельном текстовом файле.

Т И П ы и м о д у л и. Другим способом применения ТИП-технологии на языке Эль-76 является использование модульных объектов (п. 1.31). Описание интерфейса ТИПа программируется в виде описания интерфейса модуля, описание реализации ТИПа – в виде описания реализации модуля. Например:

```
интерфейс % --- ТИПа анкета --- %
% ----- ТИП анкета для обработки анкет, см. 5.4.5 ----- %
 ф64 % ----------- интерфейс доступа ----------- %
   аимя,            % (а) : имя
   агод,            % (а) : год
    . . .
   афио,            % (ф, и, о, нач) : поиск анкеты заданного лица
```

```
% ------------- интерфейс генерации ------------ %
    адайанк,          % (ск): отведение памяти для анкеты (ск слов)
    ...
    % --------- интерфейс модификации --------- %
    абрак,            % (фм, им, ом, фж, иж, ож): заключение брака
% -------------- интерфейс вывода -------------- %
    апечимя,          % (имя): вывод атрибута имя
    ...
    апечанкет         % (от, до): вывод анкет заданного диапазона
конинт % --- //интанк – имя файла объектного кода интерфейса
реал //интанк % ----- реализация ТИПа анкета ----------- %
   начало % ----- описание конкретного представления -- %
      конст дланк = 521, ...
% ---------- описание процедур интерфейса ---------- %
   процедура раимя = функция (а) % имя
                 (анк [а] . аимяп), ...
                 рапечанкет = проц (от, до) % печать анкет
                 ( ... % см. 5.4.5
                 ); % --- рапечанкет --- %
% ----------- инициализация интерфейса ----------- %
    аимя := раимя; ... ; апечанкет := рапечанкет
конец % ----- //реаланк – имя файла объектного кода
% ----------- использование ТИПа анкета----------- %
контекст конст анкета # //интанк = ( //реаланк); анкета
начало ... аимя (х) ... конец
```

Достоинство этого способа в том, что обеспечивается инкапсуляция конкретного представления и реализации операций. Кроме того, описание ТИПа в виде модуля может быть параметризовано; например, описание ТИПа *анкета* – размером таблицы.

В силу определенной последовательности развития языка и системы Эль-76 при разработке трансляторов по ТИП-технологии применялся главным образом первый метод (без модулей), а второй использовался главным образом для подключения технологических пакетов ИНТЕРФОК и ИНТЕРПАМ.

В целом отметим, что два метода описания ТИПов, рассмотренные выше, составляют удобную гибкую основу для применения ТИП-технологии на языке Эль-76 для различных задач обработки данных.

5.4.7. Использование языка Паскаль. Стандартный язык Паскаль, как уже отмечалось в гл. 2, не имеет встроенных средств модульного программирования. Поэтому при программировании по ТИП-технологии на языке Паскаль-Эльбрус для описания ТИПов используются обычные описания констант и переменных (конкретное представление) а также процедур (операции интерфейса). Тексты ТИПов хранятся в отдельных текстовых файлах и объединяются в программу с помощью библиотечных карт.

Паскаль-программа, разработанная по ТИП-технологии, должна быть оттранслирована в специальном режиме, задаваемом управляющей картой:

¤ *тип*

Этот режим имеет следующие особенности:

– в тексте программы разрешен произвольный порядок описаний;

– вызовы всех процедур и функций реализуются открытой подстановкой (т.е. по умолчанию устанавливается режим *откр*, см. гл. 2).

Необходимость произвольного порядка описаний вызвана тем, что программа состоит из нескольких ТИПов, в каждом из которых могут присутствовать любые разделы описаний, а текст каждого ТИПа хранится отдельно. Режим открытой подстановки для некоторых (наибольших по объему) процедур можно отключать, помещая в начале описания процедуры *p* прагмат ¤ *уст*0 *откр p*, а в конце ¤ *вос откр p*.

Пример. Фрагменты ТИПа *анкета*, запрограммированного на языке Паскаль в системе "Эльбрус":

```
¤ тип
(* ------ ТИП анкета: обработка анкетной информации ------ *)
(* -------- автор: В.О. Сафонов. Версия: 31.08.87 -------- *)
(* -----------конкретное представление ТИПа----------- *)
type пол = (муж, жен);                   (* пол *)
     строка = packed array               [1 .. 20] of char (* строка *)
     укстрока = ↑ строка;                (* указатель на строку *)
     уканкета = ↑ анкета;                (* указатель на анкету *)
     анкета = packed record              (* анкета *)
       имя, отч: char;                   (* инициалы *)
       год : 0 .. 2000;                  (* год рождения *)
       месяц: 1 .. 12;                   (* месяц *)
       число: 1 .. 31;                   (* число *)
       женат: boolean;                   (* признак "женат" *)
       супруг: уканкета;                 (* ссылка на супруга *)
       фам: укстрока;                    (* ссылка на фамилию *)
       след: уканкета;                   (* следующая анкета *)
     case пол: пол of                    (* пол *)
       муж: (воен: boolean;              (* воинская обязанность *)
             звание: 1 .. 50);           (* звание *)
       жен: (чисдетей: 0 .. 31)          (* число детей *)
     end (* анкета *);
var перванк: уканкета; (* ссылка на первую анкету *)
   теканк: уканкета; (* ссылка на последнюю анкету *)
(* ------- интерфейс доступа -------- *)
(* ‹‹‹‹‹‹ уровень определения (1) ››››››› *)
function аимя (а: уканкета): char;
  begin аимя := а ↑. имя end; (* аимя *)
  function афам (а: уканкета): укстрока;
    begin афам := а ↑ . фам end; (* афам *)
(* ‹‹‹‹‹‹‹‹‹‹ уровень 2: виртуальный ›››››› *)
¤ усто откр афио
function афио (var ф: строка; и, о: char; нач: уканкета): уканкета;
    var продолж: boolean;
       а, с: уканкета;
    begin продолж := true; с := нач; а := nil;
      while продолж do
```

```
            if c = nil then продолж := false
            else if (афам (c) ↑ = ф) and (аимя (c) = и) and (аотч (c) = о)
            then begin a := c; продолж := false end
            else c := аслед (c);
            афио := a
        end; (* афио *)
  ¤ вос откр афио
   (* ––––––––– интерфейс генерации ––––––– *)
   (* ⟨⟨⟨⟨⟨⟨⟨⟨⟨ уровень представления (0) ⟩⟩⟩⟩⟩⟩⟩ *)
   function адайанк (n : пол) : уканкета;
   (* размещает новую анкету *)
   var a: уканкета;
   begin new (a, пол); a ↑ . след := nil;
      if перванк = nil then перванк := a
      else теканк ↑ . след := a;
      теканк := a;
      адайанк := a
   end; (* адайанк *)
  (* ⟨⟨⟨⟨⟨⟨⟨⟨⟨ уровень определения (1) ⟩⟩⟩⟩⟩⟩⟩ *)
    ¤ уст0 откр амуж
  function амуж (ф: строка; и, о: char; г, м, ч: integer;
            ж: boolean; c : уканкета; b: boolean; з: integer) : уканкета;
     var a: уканкета;
     begin
          (* отведение памяти *)
           a := адайанк (муж);
          (* заполнение полей *)
          with a ↑ do begin
            new (фам); фам ↑ := ф;
            имя := и; отч := о; год := г;
            мес := м; число := ч; женат := ж;
            супруг := c; воен := в; звание := з
     end;
     амуж := a
  end; (* амуж *)
    ¤ вос откр амуж
          . . .
  (* ––––––––––––– конец ТИПа анкета ––––––––––––– *)
```

Как видно из примера, программирование на языке Паскаль по ТИП-технологии имеет свои особенности. Прежде всего, инкапсуляция не обеспечивается и является лишь элементом технологической дисциплины. Кроме того, Паскаль – статический язык, поэтому правильность вызова операций интерфейса в некоторой степени контролируется транслятором: например, не допускается вызов функции *афам* с аргументом типа, отличного от *уканкета*.

Однако статический контроль типов в языке Паскаль вызывает и некоторые неудобства: например, в отличие от Эль-76, для использования ТИПа *анкета*, кроме интерфейса операций, необходимо знать вспомогательные типы: *уканкета, пол, строка* и *укстрока*. Есть и менее существенные неудобства: например, необходимость представления фамилии указателем

на строку вследствие того, что функция в языке Паскаль может выдавать результат лишь простого типа или типа "указатель". Язык Паскаль не имеет текстовых макросов, поэтому во входной язык Паскаль-Эльбрус для применения ТИП-технологии введены открытые процедуры. В примере все процедуры и функции интерфейса ТИПа анкета, кроме самых больших по объему (*афио* и *амуж*), транслируются как открытые. Для использования ТИПа *анкета*, как и на языке Эль-76, необходимо подключить его к использующей программе библиотечной картой:

¤ *биб//анкета*

5.4.8. Использование языка Клу. На языке Клу описание ТИПа представляется в виде описания кластера. Конструкции языка Клу обеспечивают выражение всех компонент ТИПа – конкретного представления, интерфейса операций и их реализации – в явном и наглядном виде. Конкретное представление и реализация инкапсулированы внутри кластера. Новый класс модулей (итераторы), имеющийся в языке Клу, удобен для реализации операций интерфейса доступа концептуального уровня. Сокращенное обозначение операций модификации *store* и *set* в форме присваивания удобно для использования интерфейса модификации.

В некоторых случаях ТИП на языке Клу более удобно описывать в виде не одного, а нескольких определений кластеров. Например, удобно ТИП *анкета* рассматривать как описание *двух* абстрактных типов данных: самой анкеты и списка анкет. Операции, определенные в примере п. 5.4.5, относятся к обоим этим типам: операции выборки и модификации атрибутов (*аимя*, *аотч* и т.п.) – к анкете; операции создания, поиска и связывания анкет, например *афио*, – к списку анкет. Язык Клу позволяет описать оба типа независимо: *анкета* – отдельным кластером, *список* – отдельным параметризованным кластером, в котором тип анкеты является параметром.

При этом конкретизация АТД *список* [*анкета*] будет наиболее адекватным представлением ТИПа *анкета* на языке Клу. Описания нескольких АТД в одном кластере совмещены быть не могут.

Таким образом, в общем случае ТИП на языке Клу представляется либо в виде одного кластера, либо в виде совокупности кластеров, связанных между собой отношением параметризации. Связь между кластерами может быть выражена также через стандартные генераторы типов: например, список анкет может выражаться через генератор типа "массив":

список_анкет = **array** [*анкета*]

Но в этом случае в качестве операций над списком анкет должны использоваться встроенные операции над массивом (конструктор массива, *addh*, *addl* и т.п.), что недостаточно наглядно.

Рассмотрим пример описания ТИПа *анкета* на языке Клу в виде описания кластера с собственными переменными. При таком способе описания абстрактными объектами, подлежащими обработке, считаются анкеты, а их список (необходимый лишь для того, чтобы осуществлять доступ к анкетам и их пополнение) инкапсулирован в кластере и представлен в виде собственной переменной типа "массив конкретных представлений анкет". При этом проблема объединения нескольких АТД в одном кластере не

возникает, однако такое описание применимо только для случая, когда в Клу-программе используется не более одного экземпляра списка анкет. В противном случае следует применять метод описания ТИПа в виде двух описаний кластеров (*анкета* и *список* [*t*]). Приведены фрагменты описания.

```
анкета = cluster is
        % ––––––––– интерфейс доступа: ––––––––– %
          имя,                      % имя
          отч,                      % отчество
          . . .
          фио,                      % анкета заданного лица
          анкеты,                   % итератор анкет
        % ––––––––– интерфейс генерации ––––––– %
          муж,                      % формирование анкеты мужчины
          . . .
        % ––––––––– интерфейс модификации ––––– %
          брак,                     % внесение информации о браке
        % ––––––––– интерфейс вывода ––––––––– %
          печимя,                   % вывод имени
. . .     печанкет;                 % вывод списка анкет
супруг = variant [хол: null, жен: анкета];
мужчина = record [воен: bool, звание: int];
женщина = record [чисдетей : int];
сменная_часть = variant [муж : мужчина, жен : женщина];
rep  = record                        % конкретное представление
     [ имя, отч: char,               % инициалы
       год, месяц, число: int,       % дата рождения
       женат: bool,                  % признак "женат"
       супруг : супруг,              % ссылка на супруга
       фам: string,                  % фамилия
       вар: сменная_часть            % сменная часть
     ];  % конец конкретного представления
спис_анк = array [rep];              % тип "список анкет"
own список: спис_анк := спис_анк $ new ( );  % список анкет
% –––––––––––– реализация операций –––––––––––– %
% ––––––––––––  интерфейс доступа  –––––––––––– %
имя = proc (a: cvt) returns (char)
          return (a. имя)
        end имя;
. . .
чисдетей = proc (a : cvt) returns (int) signals (мужчина)
     if сменная_часть $ is_жен (a.вар)
     then return (сменная_часть $ value_жен (a.вар) .
                                                    чисдетей)
     else signal мужчина
     end
end чисдетей;
анкеты = iter (    ) yields (cvt)
```

```
    for a : rep in спис_анк $ elements (список) do
        yield (a)
    end
  end анкеты;
%- - - - - - - - - - интерфейс генерации - - - - - - - - - -%
  муж = proc (ф: string, и, о: char, ч, м, г: int, ж: boll,
          с : cvt, b: bool, з : int) returns (cvt)
          x : супр;
          if ж then x := супр $ make_жен (с) else x := супр $ make_хол
                                            (nil) end;
          return (rep $ [фам: ф, имя: и, отч: о, год: г, месяц: м,
          число: ч, женат: ж, супруг: х,
          вар: сменная_часть $ make_муж
                (мужчина $ [воен: в, звание : з])])
  end муж...
  end анкета;
% использование: вывод фамилий из списка анкет
    for a: анкета in анкета $ анкеты (  ) do
        анкета $ печфам (а)
    end
```

Для повышения эффективности при использовании кластера *анкета* в системе Клу-Эльбрус можно применить открытую подстановку его операций и статическую сборку программы (см. гл. 3).

5.4.9. Использование языка Модула-2. Наиболее адекватным способом представления ТИПа на языке Модула-2 является описание модуля (хотя можно применять и другой способ, предложенный в п. 5.4.7 для языка Паскаль, так как язык Модула-2 построен на базе языка Паскаль). В список импорта входят используемые элементы интерфейса других ТИПов (модулей); список экспорта задает интерфейс описываемого ТИПа, состоящий из процедур-операций и участвующих в этих операциях нестандартных типов. Для конкретного представления используются описания переменных, для реализации операций – процедуры. В разделе операторов модуля может быть указана инициализация ТИПа. Все указанные объекты, кроме списка экспорта, инкапсулированы внутри модуля.

В качестве примера рассмотрим ТИП *анкета*, запрограммированный на языках Эль-76, Паскаль и Клу в п. 5.4.4 – 5.4.8. Приведены характерные фрагменты описания, подчеркивающие основные черты языка Модула-2. Ввиду близости языка Модула-2 к языку Паскаль аналогичные конструкции и комментарии опущены.

```
module анкета; (* ТИП для обработки анкет *)
   (* используемые ТИПы: *)
   from ввод_вывод import печцел, печвещ, . . . , новстрок;
   from дин_память import allocate, deallocate;
    (* интерфейс ТИПа: *)
   export уканкета, (* тип "указатель на анкету" *)
          имя, фам, . . . , печанкет; (* операции *)
   (* конкретное представление ТИПа: *)
```

```
type пол = (муж, жен);
     строка = array [1 .. 20] of char;
     уканкета = pointer to анк; (* экспортируется *)
     анк = record
               имя, отч: char;
               год: [0 .. 2000];
               . . .
          end; (* анк *)
var перванк, теканк: уканкета;
(* реализация операций интерфейса *)
   procedure имя (a: уканкета): char;
      begin return a ↑. фам end имя;
   . . .
begin (* инициализация ТИПа *)
      перванк := nil
end  анкета;
(* использование ТИПа "анкета" *)
module социология;
   from анкета import уканкета, имя, . . . , печанкет;
   . . .
var a: уканкета;
  a := муж ("иванов", "и", "и", 1960, 12,5, false, nil, true, 1);
печанкеты (a)
  . . .
end социология
```

Понятие модуля в языке Модула-2 адекватно отражает как саму концепцию ТИПа, так и ряд принципов ТИП-технологии: инкапсуляцию, связи между ТИПами, описания их интерфейса и реализации (последние особенно наглядно представляются в виде раздельных описаний: **definition module** – интерфейс, **implementation** – реализация). С целью повышения эффективности исполнения операций при программировании по ТИП-технологии на языке Модула-2 в системе Модула2-Эльбрус предусмотрены режимы открытой подстановки процедур (¤ *уст откр*) и статической сборки программы (¤ *сборка //м*). Язык Модула-2 статически типизирован, но содержит и ограниченные элементы динамизма: предопределенный тип *word* (слово – аналог описателя **ф64** в языке Эль-76). С помощью этого типа любую структуру данных – фактический параметр некоторой операции – можно рассматривать как массив слов (**array of** *word*) и, тем самым, в реализации операции иметь непосредственный доступ к конкретному представлению. Аналогично предопределенный тип *address* = **pointer to** *word* позволяет отключать контроль типов при передаче в операцию указателя на структуру данных. Такой стиль программирования близок к языку Эль-76. Контроль правильности реализации (корректности доступа к элементам представления) возлагается на программиста и частично обеспечивается аппаратурой системы "Эльбрус" во время исполнения программы.

5.5. Применение ТИП-технологии в трансляторах

Принципы ТИП-технологии, описанные в § 5.3–5.4, успешно применены при разработке систем программирования Паскаль-Эльбрус, Клу-Эльбрус, АБВ-Эльбрус и Модула-2-Эльбрус. В этом параграфе рассматриваются методы применения ТИП-технологии при разработке трансляторов, анализируются опыт и результаты ее использования.

5.5.1. Структура системы программирования. Разработка трансляторов – один из наиболее традиционных видов деятельности системных программистов. Кроме того, перечисленные системы программирования для МВК "Эльбрус", несмотря на различия входных языков, имеют между собой много общего по выполняемым функциям, используемым структурам данных и программным компонентам. Ввиду всего этого первоначальный этап конструирования программы по ТИП-технологии (разработка общей структуры системы в виде совокупности ТИПов), который весьма сложен для малоизученной предметной области, при создании систем программирования значительно упрощается.

Система программирования, как правило, создается в виде совокупности следующих ТИПов:

– *ввод-вывод* – ТИП, определяющий общий интерфейс ввода-вывода уровня представления во время трансляции. Он используется во всех транслятора. Под уровнем представления здесь понимается вывод в терминах элементарных типов данных целевой ЭВМ: *печцел* (*целое*), *печстр* (*строка*) и т.п.;

– *лексический анализ* – ТИП, описание которого состоит из глобальных констант (классов лексем), интерфейсных переменных (выходной информации сканера) и собственно лексического анализатора. ТИП "лексический анализ" содержит локальный ТИП "*обработка управляющих карт*". Используется ТИПами синтаксического анализа;

– *синтаксический и семантический анализ* – совокупность ТИПов для анализа различных групп конструкций входного языка. Обычно выделяются ТИПы для анализа обозначений ТИПов (видов), описаний, переменных, выражений и операторов;

– *работа с таблицами* – совокупность ТИПов, обеспечивающая интерфейс для работы с таблицами транслятора при лексическом синтаксическом и семантическом анализе. Три основных ТИПа этой группы предназначены для работы с таблицей внешних представлений идентификаторов ТАБЛИД, информационной таблицей описаний ИНФО и таблицей обозначений типов ТИПЫ. В зависимости от входного языка и особенностей реализации могут быть запрограммированы дополнительные ТИПы для работы с таблицами меток, ситуаций и т.п.;

– *сервис* – совокупность ТИПов для выполнения сервисных функций системы программирования – диагностики ошибок, реализации режимов, управляемых пользовательскими прагматами, трассировки, инициализации и завершения работы транслятора;

– *генерация кода* – ТИП для генерации команд целевой ЭВМ. В системе "Эльбрус" эту функцию выполняет технологический пакет ИНТЕРКОД [8], входящий в состав ОСПО МВК "Эльбрус";

– *генерация файла объектного кода* – ТИП для генерации стандартной структуры файла объектного кода, системных таблиц и другой выходной информации компилятора. На МВК "Эльбрус" выполнение технически весьма трудоемких действий по генерации расширенного файла объектного кода возложено на технологический пакет ИНТЕРФОК, входящий в ОСПО МВК "Эльбрус" и реализованный как модульный объект;

– *динамическая поддержка* – ТИП для выполнения необходимых функций системы программирования во время счета: ввода-вывода, предопределенных процедур, распределений памяти для динамических объектов, диагностики динамических ошибок. В системе "Эльбрус" ТИП для динамической поддержки целесообразно программировать в виде описания модуля и подключать во время исполнения программы вызовом генератора модульного объекта, например: *ген* (*//реалдин*), где *//реалдин* – внешнее имя файла объектного кода реализации модуля динамической поддержки.

Многие из перечисленных ТИПов могут без существенных изменений использоваться в других системах программирования и составляют инструментарий нижнего уровня для разработки систем программирования на МВК "Эльбрус". При этом могут потребоваться изменения в следующих ТИПах: в ТИПе "лексический анализ" – замена отдельных лексических классов: в ТИПе "выражение" – замена отдельных видов выражений; в ТИПах для работы с таблицами – замена отдельных классов элементов таблиц; в ТИПах "переменная", "оператор", "описание" – замена отдельных видов переменных, операторов, описаний. Остальные ТИПы играют роль дополнительных концептуальных средств, облегчающих разработку других систем программирования. Ввиду единого систематического подхода к разработке всех трансляторов в новых системах программирования возможно использование отдельных модулей из других ТИПов. Например, текст системы программирования Модула-2-Эльбрус получен из текста системы программирования Паскаль-Эльбрус путем замены и дополнения некоторых модулей и ТИПов.

5.5.2. Работа с таблицами. Для каждой таблицы транслятора определен ТИП с трехуровневым интерфейсом доступа, генерации, модификации и вывода. Организация ТИПов для работы с таблицами транслятора имеет свою специфику, связанную с тем, что каждая из них является представлением какого-либо класса конструкций входного языка. Таблица состоит из элементов; каждый элемент таблицы имеет свой *прообраз* во входном тексте программы. Например, прообразом элемента таблицы идентификаторов ТАБЛИД является внешнее представление идентификатора, таблицы описаний ИНФО – описание объекта (константы, переменной, процедуры и т.п.), таблицы обозначений типов ТИПЫ – обозначение типа (предопределенный тип, тип-перечисление, тип-массив и т.п.). Каждая таблица представлена в виде вектора переменной длины; элемент таблицы занимает целое число слов и характеризуется индексом начального слова. Для классификации элементов таблицы среди полей (атрибутов) элемента выделено одно поле – *класс* (*категория, сорт* и т.п.), играющее роль поля признака.

В соответствии с описанной структурой таблиц, их представлением и назначением уровни абстракции ТИПа для работы с каждой таблицей реализуются следующим образом.

Уровень представления – описание конкретного представления таблицы в виде описаний констант, переменных и полей; описание интерфейса генерации для управления памятью (см. пример п. 5.4.5). Пример модуля уровня представления интерфейса генерации – *идайинфо* (*ск*) – размещение элемента таблицы ИНФО длиной *ск* слов. Интерфейс вывода уровня представления является общим для всех трансляторов.

Уровень определения – уровень работы в терминах полей элемента таблицы. Пример модуля интерфейса доступа: *икат*(*и*) → *кат* (категория элемента таблицы ИНФО), модуля интерфейса генерации – *ипеpемпоз* (*ид, ур, смещ, тип*) → *и* (формирование элемента таблицы ИНФО для описания переменной; результат – ссылка в ИНФО), модуля интерфейса вывода – *печид* (*ид*) (вывод идентификатора). Смысл параметров: *и* – индекс в таблице ИНФО, *ид* – идентификатор (ссылка в ТАБЛИД), *ур* – лексикографический уровень переменной, *смещ* – смещение переменной, *тип* – тип переменной (ссылка в ТИПы).

Концептуальный уровень – уровень работы в терминах прообразов элементов таблицы. В качестве примера рассмотрим таблицу обозначений типов. Элемент таблицы, прообразом которого является обозначение типа "запись", должен содержать информацию о полях записи. Поскольку для каждого поля записи в таблице имеется отдельный элемент и, следовательно, для получения информации о поле записи с заданным именем требуется ассоциативный поиск соответствующего элемента, то наиболее технологичным решением будет включение в ТИП модуля интерфейса доступа концептуального уровня, выполняющего этот поиск, со следующей спецификацией:

тполе(*т, имя*) → *поле* или 0 ,

где *т* – индекс элемента-образа типа "запись", *имя* – индекс имени поля записи в ТАБЛИД, *поле* – индекс элемента-образа поля записи. Результат 0 выдается при неудачном поиске.

Интерфейс генерации концептуального уровня для обработки таблиц проектируется в соответствии с этапами синтаксического анализа прообраза, т.е. с его абстрактным синтаксисом. В качестве примера рассмотрим интерфейс генерации элемента таблицы типов в системе Клу-Эльбрус, прообразом которого является обозначение процедурного или итераторного типа (п. 3.3.8). Напомним синтаксис процедурного типа в языке Клу:

① **proctype** (. . . a_i, ② . . .) ③ **returns** (. . . r_j, ④ . . .) ⑤
signals (. . . s_k ⑥ (. . . t_{kl}, ⑦ . . .), ⑧ . . .) ⑨

Здесь: a_i – типы аргументов; r_j – типы результатов; s_k – имена ситуаций; t_{kl} – типы аргументов ситуации s_k. Цифрами 1–9 помечены элементы синтаксической структуры и соответствующие им этапы синтаксического анализа. Каждому из этапов ставится в соответствие модуль интерфейса генерации, выполняющий частичную генерацию элемента таблицы обозначений типов:

1) *тпроцтипнач* (*@ аргсит*) → *буф* – начало обозначения типа;
2) *тпроцтипарг* (*типарг, номарг*) – занесение типа аргумента;
3) *тпроцтипконарг* (*чисарг*) – занесение числа аргументов;

4) *троцтипрез* (*типрез, номрез*) – занесение типа результата;

5) *троцтипконрез* (*чисрез*) – конец результатов (занесение числа аргументов и результатов);

6) *троцтипначсит* (*имясит, номсит*) – занесение имени ситуации;

7) *троцтипаргсит* (*номсит, аргсит, типаргсит, номаргсит*) – занесение типа аргумента ситуации;

8) *троцтипконсит* (*номсит, чисаргсит*) – конец ситуации (занесение числа аргументов ситуации);

9) *троцтипкон* (*нов, сорт, буф, чиссит, аргсит*) → *т* – конец обозначения типа; сортировка ситуаций по алфавиту их имен; поиск элемента в таблице обозначений типов; при неудачном поиске – занесение элемента в таблицу.

Здесь: *типарг* – тип аргумента; *номарг* – номер аргумента; *чисарг* – число аргументов (аналогично – для результатов и ситуаций); *имясит* – имя ситуации (индекс в ТАБЛИД); *буф* – индекс начала элемента в буфере; *аргсит* – переменная для вспомогательного массива аргументов ситуаций (массив создается модулем *процтипнач* и освобождается модулем *процтипкон*); *нов* – эвристика для поиска (если значение *нов* истинно, то поиск элемента в таблице типов не выполняется, и он сразу заносится в таблицу); *сорт* – сорт (класс) типа: процедурный или итераторный тип (последний имеет аналогичную структуру); *т* – индекс элемента в таблице типов.

Поскольку обозначения типов, в соответствии с семаниткой языка Клу, должны проверяться на структурную идентичность, элемент-образ обозначения сложного типа формируется сначала во вспомогательном буфере, а затем, в случае неудачного поиска идентичного элемента, переписывается в таблицу типов. Все эти действия локализованы в модуле *процтипкон*.

Интерфейс вывода концептуального уровня обеспечивает вывод табличной информации в форме, максимально близкой к исходному тексту прообраза элемента (т.е. синтез прообраза по элементу таблицы). Например, если элемент таблицы ИНФО в системе Паскаль-Эльбрус с индексом *и* соответствует описанию переменной *а*, то модуль *ипечперем* (*и*) выводит информацию об этом описании переменной в форме

```
var a: integer
```

Содержимое элемента таблицы обозначений типов *т* в системе Клу-Эльбрус, соответствующего типу "запись" с полями *x* типа **real** и *y* типа **array** [**char**], выводится модулем *печтип* (*т*) в форме

```
record[x : real, y : array [char]]
```

5.5.3. Синтаксический анализ. ТИПы для синтаксического анализа конструкций исходного текста конструируются по тем же принципам структурирования интерфейса. Например, анализатор выражений в системах Паскаль-Эльбрус и Клу-Эльбрус [17] запрограммирован как ТИП. Его интерфейсом доступа является группа модулей анализа компонент выражения, интерфейсом генерации – модули генерации внутреннего представления выражения (стек МП-автомата), интерфейсом вывода – модули генерации кода и трассировки работы анализатора.

Подробнее рассмотрим более простой пример – ТИП для анализа описаний в системе Паскаль-Эльбрус. Синтаксическая структура описаний языка

Паскаль строится из следующих элементов:

– обязательная лексема (например, **array**);

– список идентификаторов (например, *а, в, с*);

– список произвольных конструкций с разделителем "," или ";". Например, в обозначении типа "запись":

record *a, b, c*: *integer*; *d*: *real* **end**

встречаются все три вида этих синтаксических элементов. В соответствии с этим интерфейс доступа (анализа) концептуального уровня состоит из модулей:

съел (*символ*) – анализ обязательного спецсимвола;

списокид (*условиеконца, семантика*) – анализ списка идентификаторов;

список (*условиеконца, конструкция, разделитель, номерошибки, семантика*) – анализ списка произвольных конструкций с произвольным разделителем.

Здесь: *символ* – лексический класс символа; *условиеконца* – множество лексем, которые могут ограничивать список; *семантика* – семантическая подпрограмма для вхождения конструкции; *конструкция* – синтаксический анализ конструкции; *разделитель* – лексический класс разделителя; *номерошибки* – номер ошибки в списке конструкций (для диагностики).

5.5.4. Семантический анализ и контроль ТИПов основаны на использовании ТИПов для работы с таблицами. Наиболее часто встречающиеся семантические проверки целесообразно оформить в виде отдельных модулей, которые по принятой классификации можно отнести к интерфейсу доступа уровня 2 соответствующего ТИПа. Например, *тсовмтипы* (*т*1, *т*2) – проверка на совместимость типов *т*1 и *т*2, *тэтопростип* (*т*) – проверка на принадлежность типа *т* к простым типам.

5.5.5. Опыт и результаты. На основе ТИП-технологии разработано четыре системы программирования для МВК "Эльбрус" (Паскаль-Эльбрус, Клу-Эльбрус, АБВ-Эльбрус, Модула-2-Эльбрус). При программировании использовался базовый язык Эль-76. Производительность труда при разработке составила в среднем порядка 10 тысяч строк программ в год, а при разработке систем Паскаль-Эльбрус и Клу-Эльбрус – около 20 тысяч строк в год на одного программиста (от начала разработки до конца отладки).

Применение ТИП-технологии значительно облегчило конструирование программ на всех этапах жизненного цикла.

1. *Спецификация* модулей ТИПа достаточно проста ввиду их небольшого объема и ограниченных информационных связей.

2. *Проектирование* модулей ТИПа облегчается благодаря систематическому применению метода восходящей разработки и концептуальным средствам для классификации и слоения интерфейса.

3. *Программирование* существенно упрощено благодаря простоте и обозримости модулей и наличию уже реализованных примитивов нижнего уровня. Процесс составления программ и мышление программиста систематизируются. ТИП последовательно конструируется из модулей интерфейса, а программа – из ТИПов, как из готовых строительных блоков. Значитель-

ная часть ТИПов и их отдельных модулей может быть использована в других трансляторах. Опыт показывает, что вполне реальна задача создания настраиваемого набора ТИПов для автоматизации разработки программ в конкретной предметной области.

4. *Отладка программы* сводится к контролю правильности обращения к модулям интерфейса. Число ошибок в реализации самих модулей очень мало, так как ввиду их простоты и небольшого объема их "ручная" верификация легко может быть выполнена в процессе программирования. Обнаруженные ошибки можно классифицировать следующим образом:

– ошибки, связанные с неверным типом или смыслом фактического параметра модуля;

– ошибки, связанные с неверным порядком параметров (сказываются неудобства позиционного способа передачи параметров, особенно при большом их числе);

– ошибки вида "плюс-минус единица" при передаче целочисленных параметров (индексов).

Благодаря декомпозиции программы на ТИПы эти ошибки легко локализуются, обнаруживаются и устраняются.

5. *Сопровождение и модификация программы* также выполняется в соответствии с ее декомпозицией на ТИПы, что систематизирует и облегчает работу. Любое изменение в программе выражается либо в добавлении новых ТИПов, составляющих новый вертикальный слой, либо локализуется на определенном уровне интерфейса ТИПа. При этом модификации использующих модулей, как правило, не требуется.

Наиболее сложные этапы в ТИП-технологии – проектирование общей структуры системы и проектирования интерфейса. Соблюдение интерфейса также вызывает некоторые трудности психологического порядка, которые преодолеваются с опытом коллективной разработки программ.

5.6. Развитие ТИП-технологии

В заключение главы рассмотрим направления развития и перспективы ТИП-технологии.

5.6.1. Автоматизация разработки ТИПов. Декомпозиция программы на ТИПы и ТИПов на уровни и виды интерфейса позволяет выделить рутинную часть работы по проектированию и реализации модулей интерфейса, которая может быть выполнена автоматически, по крайней мере, для объектов, имеющих представление либо в виде набора простых переменных (будем называть их *простыми объектами*), либо в виде таблицы (*табличных объектов*):

– построение элементов конкретного представления: описаний переменных и массивов для табличных объектов (по заданным размерам таблицы); описаний полей элемента таблицы по заданным именам и типам (размерам) атрибутов элемента;

– уровень представления интерфейса генерации для табличных объектов (если он не обеспечивается инструментальным языком);

– уровень определения интерфейса доступа, генерации и вывода для простых и табличных объектов;

– стандартные комментарии (спецификации) к элементам представления и модулям.

Для дальнейшей автоматизации разработки ТИПов (включая концептуальный уровень) требуется формализация и выделение типичных ситуаций.

5.6.2. Технологический комплекс ТИП. На основе выработанных технологических принципов и накопленного опыта разработки программ создается технологический комплекс ТИП для поддержки разработки программ по ТИП-технологии программирования в диалоговом режиме. В его основу положены следующие принципы:

– наиболее адекватная форма представления информации о ТИПе при разработке программ в диалоге – многооконное представление на экране дисплея (рис. 2);

– обеспечивается автоматизация этапов проектирования и разработки программ, описанных в п. 5.6.1, в частности, синтез части модулей ТИПа по информации, полученной из диалога. Наиболее распространенные классы объектов и атрибутов имеют стандартные обозначения: *прост, таблица, индекс, строка, цел, вещ, лог, символ, вектор*;

– остальные этапы проектирования и разработки программ поддерживаются типовыми сценариями для декомпозиции на ТИПы, уровни и виды интерфейса; в сценариях используются средства подсказки и обучения;

– контролируется соблюдение технологической дисциплины: инкапсуляции конкретного представления, соответствия уровней определения ТИПа, ограничений на использование имен; последовательности разработки

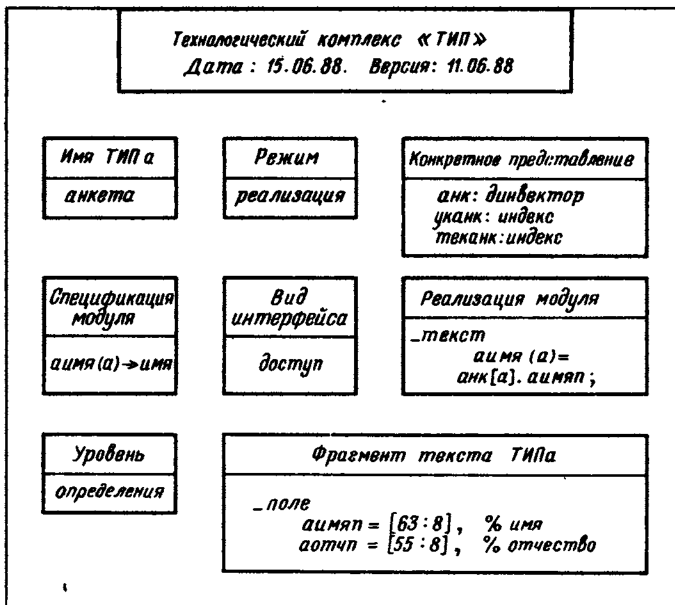


Рис. 2

снизу вверх; наличия определений интерфейса и реализации у используемых модулей ТИПа;

– результаты проектирования и разработки программ по ТИП-технологии хранятся в базе данных проекта, могут быть просмотрены и откорректированы разработчиком; программа и отдельные ТИПы, полученные в результате разработки, записываются в архив;

– недостающая для функционирования технологического комплекса информация (имена и типы модулей, их аргументов, результатов, классы объектов и т.п.) запрашивается у разработчика в диалоге и использованием подсказок и меню.

5.6.3. Перспективы ТИП-технологии. Принципы и программная поддержка ТИП-технологии, рассмотренные в данной главе, могут быть применены для конструирования программ, связанных с обработкой сложных структур данных. Прежде всего, по-видимому, ТИП-технология может оказаться полезной в следующих областях:

– *трансляторы* (опыт применения [65, 68] подтверждает ее перспективность);

– *пакеты прикладных программ* (ППП); современная трактовка ППП близка концепции ТИПа, в котором программисту предоставляется концептуальный уровень интерфейса (обращение к нему осуществляется в терминах входного языка пакета);

– *информационные системы и СУБД*; принципы инкапсуляции и виртуализации данных, применяемые в ТИП-технологии, согласуются с принципами разработки СУБД (в частности, понятие концептуального уровня интерфейса доступа близко понятию запроса к базе данных);

– *системы управления различными объектами*; модули управления удобно оформлять в виде ТИПа, реализация уровня представления которого может быть выполнена аппаратно;

– *системы обработки знаний* (*экспертные системы*) [78]. Структура экспертной системы и информационные связи ее частей в настоящее время носят достаточно устоявшийся характер (система состоит из *диалогового процессора, базы знаний, подсистемы логического вывода* и *подсистемы объяснения*). Каждая из этих частей может быть оформлена в виде ТИПа, что позволяет фиксировать информационные связи, обеспечивать эффективность модулей нижнего уровня (что весьма важно для ускорения поиска в базе знаний и унификации при логическом выводе) и выделять модули, которые могут быть использованы в других экспертных системах.

Перспективность ТИП-технологии в рассмотренных областях, кроме первой, еще предстоит доказать на практике. Но уже на основании нынешнего опыта ее использования можно сделать некоторые общие выводы.

ТИП-технология не претендует на принципиальную новизну; она является одним из практических подходов к внедрению новых методов программирования: объектно-ориентированного подхода, модульного программирования и абстрактных типов данных. Все ее принципы взяты из опыта разработки больших программ и направлены на улучшение их качества, уменьшение дублирования, повышение надежности и наглядности, устранение типичных ошибок.

Принципы ТИП-технологии не противоречат никаким из существующих технологий (Р-технологии, технологии вертикального слоения, типовой

технологии разработки программ для МВК "Эльбрус" и другим), а лишь дополняют их.

Концепции и методы ТИП-технологии близки распространенным идеям *сборочного программирования* – автоматизированного конструирования программы из готовых "строительных блоков" с постоянным повышением уровня абстракции используемых программ. Можно надеяться, что ТИП-технология будет способствовать осуществлению этих перспективных идей.

ГЛАВА 6

МЕТОДЫ РЕАЛИЗАЦИИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ В СИСТЕМЕ "ЭЛЬБРУС"

Данная глава отличается от предыдущих тем, что система "Эльбрус" описывается в ней с другой точки зрения. В гл. 1–5 основное внимание уделено внешним (пользовательским) аспектам системы: используемым языкам и методам программирования, возможностям систем программирования Эль-76, Паскаль, Клу и др. – иначе говоря, языковому *интерфейсу* системы "Эльбрус". В данной главе система "Эльбрус" рассматривается "изнутри": показано, какие именно черты ее архитектуры (*реализации*) и каким образом способствуют удобству и эффективности реализации языков программирования различных классов. В заключение главы рассматриваются новые методы реализации языков статического и динамического класса, примененные в системах программирования для МВК "Эльбрус" и основанные на использовании встроенных аппаратных средств поддержки языков высокого уровня.

Материал главы предназначен для системных программистов и может служить для ознакомления с архитектурой системы "Эльбрус" или ее начального изучения. Перед чтением данной главы рекомендуется внимательно изучить введение.

6.1. Архитектура МВК "Эльбрус" с точки зрения разработчика трансляторов

При проектировании системы "Эльбрус" в ее основу была положена ориентация на процедурные алголоподобные языки высокого уровня. Принципы этого нового подхода к архитектуре ЭВМ изложены во введении. Система команд МВК "Эльбрус", структура процессоров, организация памяти спроектированы с таким расчетом, чтобы облегчить реализацию языков, частично или полностью переложив на аппаратуру и ОС решение рутинных задач, возникающих при разработке любой системы программирования:

– лексический анализ;

– представление базовых типов данных (чисел, символов, строк, векторов);

– распределение регистров для вычисления выражений;

– реализацию процедур, размещение и адресацию их локальных величин;

– динамическое распределение памяти для языковых структур данных;

– реализацию управляющих конструкций (условнях предложений, циклов, меток и переходов, ситуаций);

– связывание независимых программ и модулей для их совместного исполнения.

Фундаментальные концепции и механизмы, "вложенные" в аппаратуру МВК "Эльбрус" для реализации перечисленных функций (тег, процедура, массив, стек и другие), оказались настолько общими, а сама архитектура МВК "Эльбрус" – столь гибкой и динамичной, что для системы "Эльбрус" удалось эффективно реализовать значительно более широкий класс языков, в том числе динамические (интерпретационные) языки, языки модульного программирования, языки с АТД.

Рассмотрим основные понятия и конструкции языков процедурного и динамического класса и соответствующие им элементы архитектуры системы "Эльбрус". При описании команд МВК "Эльбрус" используется стандартная мнемоника команд МВК "Эльбрус-2". Каждая команда изображается в виде: *код операции* (*операнды*). Последовательное выполнение команд обозначается символом " ; ".

6.1.1. Процедурные языки. Подробно определение и свойства этого класса языков рассмотрены в В. 1.2. Процедурные языки – это наиболее широко используемые языки Фортран, Алгол, ПЛ/1, Паскаль и др. (для этого класса языков используется также термин *операторные языки,* так как программа в них строится из операторов). Для каждой языковой конструкции либо указаны конкретные машинные команды МВК "Эльбрус", в которые она транслируется, либо описан метод реализации.

К о н с т а н т ы – изображения значений простых типов данных, например: 1, 0.25, "А". Каждому языковому простому типу соответствует определенный машинный тип данных МВК "Эльбрус"; вхождения констант транслируются в команды загрузки константы в *область выражений,* в которой размещаются промежуточные результаты вычислений. Константа хранится в команде как непосредственный операнд (длина команды переменная, от 1 до 10 байтов). Таким образом, в МВК "Эльбрус" не требуется организации специальной области литералов (простых констант), как в традиционных ЭВМ. При этом, по сравнению с традиционной схемой реализации констант с помощью области литералов, несколько увеличивается длина кода, но уменьшается время выборки (за счет конвейерной обработки команд и их хранения непосредственно в физической памяти).

В табл. 2 приведены обозначения простых типов данных языка Паскаль-Эльбрус (см. гл. 2), их смысл, примеры констант, соответствующий машинный тип данных и команды загрузки или формирования констант.

Мнемоника команд: *зг1* – загрузка единицы; *унзг* – универсальная непосредственная загрузка; *вэлн* – взять элемент набора непосредственно; *впнн* – взять поднабор непосредственно: *пудв* – преобразование в вещественное удвоенной точности.

Константы-строки и *структурные константы* реализуются другим способом. Элементы структурной константы при исполнении программы размещаются в векторе, защищенном по записи. Область памяти для этого вектора выделяется из особой части математической памяти, общей для всех

Таблица 2. **Реализация простых типов данных**

Языковой тип	Смысл	Примеры констант	Машинный тип	Реализация (команды)	Смысл
integer	цело число	0 1 25 1000 123456	целое 64	*зг0* *зг1* *зг8* (25) *зг16* (1000) *зг32* (123456)	загрузка нуля, единицы или целого числа не длине 8 (16, 32) двоичных разрядов
shortint	короткое целое число	5	целое 32	*зг8* (5); *ф32окр*	загрузка целого 64 и его преобразование к формату ф32
real	вещественное число	1.57	вещественное 64	*унзг* (*вещ64*, 1.57)	универсальная загрузка с указанием типа и значения
shortreal	короткое вещественное	0.5	вещественное 32	*згв32* (0.5)	загрузка вещественного 32
longreal	вещественное двойной точности	1e200	вещественное 128	*унзг* (*наб*, *ст*); *унзг* (*наб*, *мл*); *пудв*	загрузка старших и младших разрядов отдельно
boolean	логическое значение	*true* *false*	набор длины 1	*зг1*; *взлн* (0) *зг0*; *взлн* (0)	загрузка целого нуля или единицы и выделение младшего бита
char	символ	*'a'*	набор длины 8	*зг8* (*ord*(*c*)); *впнн* (7, 8)	загрузка кода и выделение младшего байта

задач. Прообраз структурной константы – значения элементов и ссылка на них (*базовый дескриптор массива констант* – БДМК) – хранится в файле объектного кода: элементы – в массиве констант, БДМК – в собственном контексте программы (СКП), области памяти для хранения ссылок на постоянные глобальные объекты. При параллельном исполнении объектного кода одной и той же программы в нескольких задачах структурная константа будет храниться в памяти в единственном экземпляре, что позволяет экономить память под массивы констант, которые могут быть весьма велики (например, постоянные таблицы транслятора). В объектном коде структурная константа представляется командами чтения БДМК из СКП и формирования дескриптора на подмассив, содержащий элементы данной структурной константы.

П е р е м е н н ы е. Каждая переменная в процедурном языке программирования локализована в некоторой *процедуре* (блоке). В реализа-

ции на традиционных ЭВМ для размещения локальных данных процедур отводится область памяти под *стек*, действия над которым реализуются подпрограммами системы динамической поддержки или последовательностью обычных команд работы с памятью. В системе "Эльбрус" классический метод реализации процедур с помощью стека вложен в аппаратуру. Все действия над стеком локальных данных выполняются аппаратно: либо неявно (при входе в процедуру и выходе из процедуры), либо специальными командами размещения переменных, взятия их значений и формирования *имен* (адресов). Область памяти для стека отводится операционной системой при создании процесса (п. 1.30.2); каждому процессу соответствует свой стек.

При вызове процедуры для размещения ее фактических параметров и локальных переменных отводится текущая часть стека – *область локальных данных* (см. рис. 3). Она начинается двумя словами системной информации для организации процедурного механизма – *маркером стека* (МКС) и *управляющим словом возврата* (УСВ). Их назначение разъясняется несколько позже. *Смещение* переменной (относительный адрес в области локальных данных) вычисляется в *полусловах*; таким образом, смещение первой переменной равно 4. Другой характеристикой переменной является ее *уровень* – число объемлющих процедур, включая программу и процедуру, в которой локализована переменная. Уровень глобальных переменных программы равен 1. Уровень 0 зарезервирован для собственного контекста программы (все значения глобальных уровней даны для МВК "Эльбрус-2"). Уровень *ур* и смещение *смещ* составляют *адресную пару* (*ур, смещ*) – однозначную характеристику переменной для заданной вложенности процедур, вычисляемую компилятором и указываемую в командах обращения к переменной. Дескриптор области локальных данных процедуры уровня *ур* при ее исполнении хранится в специальном базовом регистре и используется для обращения к локальным переменным. Всего в аппаратуре имеются 32 базовых регистра.

На рис. 3 изображено состояние стека локальных данных и базовых регистров при исполнении процедуры *p* уровня 2:

```
procedure p;
var x: integer; y : real;
begin x := 1; y := 3.14
end (* p *)
```

Переменная *x* имеет адресную пару (2, 4), переменная *y* – (2, 6). Базовый регистр уровня 2 содержит дескриптор области локальных данных (формат – ф32, длина – 8). *рг – регистр границы* между областью локальных данных и областью выражений.

Рассмотрим более подробно реализацию трех основных действий над переменными – описания, использования и присваивания.

Описание переменной. Описанию новой переменной соответствует увеличение области локальных данных (передвижение указателя стека) на одно слово (для переменных, значение которых занимает 32 разряда, – на полуслово, 128 разрядов – на два слова). Наиболее простой вид описаний – описания без инициализации. В системе "Эльбрус" значения не инициализированных переменных представляются пустыми объектами фор-

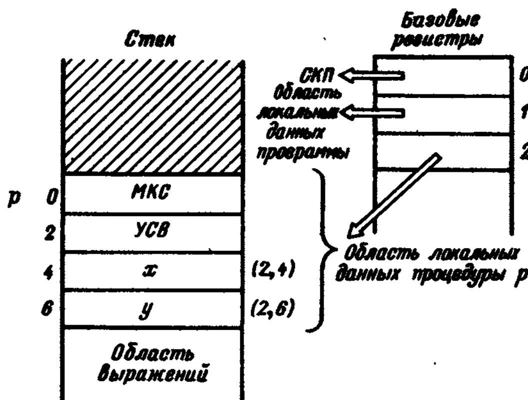


Рис. 3

матов ф32 или ф64 (см. п. 1.2). Наиболее распространенный случай – описание одной или нескольких простых переменных без инициализации – реализуется командой *пус* (*к*), где *к* ≤ 15 – число простых переменных формата ф64 (*пус* – передвинуть указатель стека). Область локальных данных увеличивается на *к* слов; переменные инициализируются пустыми объектами формата ф64. Например, описание переменных *x* и *y* в процедуре *p* реализуется одной командой *пус* (2).

Описания переменной с инициализацией:

real *x* : = 1.0,   *y* := *a* + *b*;'

реализуются следующим образом. Начальное значение каждой переменной, полученное при вычислении выражения, помещается в область выражений. Команда *пус* (0) присоединяет содержимое области выражений к области локальных данных. Приведенный фрагмент описаний на Алголе-68 реализуется командами

унзг (*вещ64*, 1.0) ; ⟨ вычисление *а* + *в* ⟩; *пус* (0) .

Использование переменной. Использующие вхождения переменной *a* с адресной парой (*ур, смещ*) транслируются в команду *вел* (*ур, смещ*), которая считывает текущее значение переменной и помещает его в область выражений (*вел* – считать величину). Для локальных переменных текущей (самой внутренней) процедуры используется более короткая команда *лвел* (*смещ*), эквивалентная *вел* (*текур, смещ*), где *текур* – уровень текущей процедуры.

Присваивание переменной a : = *e* состоит из трех действий: вычисления выражения *e*, формирования имени переменной *a* и записи значения *e* в область памяти переменной *a*. Имя переменной *a* формируется и загружается в область выражений командой *зга* (*ур, смещ*) либо *лзга* (*смещ*), если *a* – локальная переменная текущей процедуры. Мнемоника: *зга* – загрузка адреса, *лзга* – загрузка адреса локальной переменной.

Для записи (присваивания) значения переменной в память предусмотрены несколько вариантов команд. Наиболее часто используется команда

зб4 – запись значения не длиннее 64 разрядов. Для ее выполнения в область выражений должны быть загружены значение выражения *e* и имя переменной *a* (именно в такой последовательности). Команда *зб4* вычеркивает операнды из области выражений и выполняет присваивание переменной с проверкой формата значения (если он превышает **ф64**, то преобразуется к этому формату; если это невозможно – прерывание). Таким образом, оператор

a := 1 ;

реализуется последовательностью команд

зг1; *зга* (*ур, смещ*); *зб4*

причем команды *зга* и *зб4* воспринимаются аппаратурой как *одна* команда (аппаратно оптимизируются). Именно с этой оптимизацией связано изменение порядка операндов и использование команды *зб4* при генерации кода присваивания простой переменной.

Переменные-записи (*структуры*), поля которых имеют простой тип или вновь являются записями, реализуются теми же командами, что и простые переменные. В системе Паскаль-Эльбрус (см. гл. 2) такие записи называются *прямоадресуемыми*. Поля прямоадресуемых записей хранятся в области локальных данных и адресуются с помощью адресных пар. Например, следующее описание переменной *z* в процедуре *q*:

procedure *q* ; (* *уровень* = 2 *)
var *z* : **record** *re, im* : *real* **end** ; . . .

реализуется командой *нус* (2). Полю *z . re* соответствует адресная пара (2, 4), полю *z . im* – адресная пара (2, 6). Оператор:

z . re := 0.0

транслируется в последовательность команд:

унзг (*вещ64*, 0.0) ; *зга* (2, 4) ; *зб4*.

В ы р а ж е н и я. Традиционным методом реализации выражений в языках программирования является их перевод в обратную польскую запись и последующее распределение регистров для их вычисления. Оба эти этапа реализации выполняются компилятором. В системе "Эльбрус" (в моделях МВК "Эльбрус-1" и МВК "Эльбрус-2") второй этап (распределение регистров) выполняется аппаратно при исполнении программы. С точки зрения системного программиста – разработчика транслятора, в системе "Эльбрус" для вычисления выражений имеется *стек* (*область выражений*). Система команд представляет собой обратную польскую запись выражений. Например, пусть в процедуре *p*, область локальных данных которой изображена на рис. 3, вычисляется выражение $x + y * x - 1$. Обратная польская запись этого выражения: $xyx * + 1 -$. Выражение транслируется в последовательность команд:

вел (2, 4) ; *вел* (2, 6) ; *вел* (2, 4) ; *умн*; *сл*; *зг1*; *вчт*,

которая получается заменой в обратной польской записи вхождений переменных командами *вел*, констант – командами их загрузки и знаков

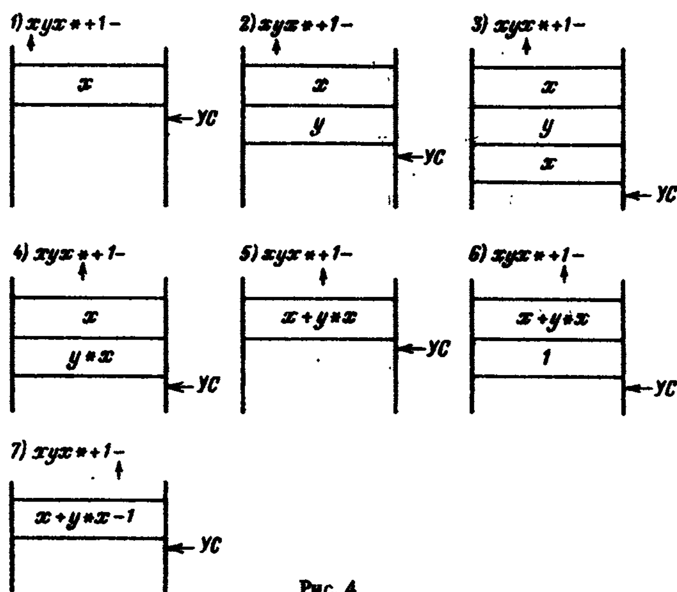


Рис. 4

операций – командами, выполняющими соответствующие арифметические операции.

Каждая команда, реализующая некоторую операцию (например, *сл* – сложение) берет операнды из вершины стека выражений, вычеркивает их и помещает в вершину стека результат. Последовательность шагов вычисления выражения и соответствующих им состояний области выражений изображена на рис. 4. Результат вычисления выражения остается на вершине стека.

Отметим еще две особенности системы команд МВК "Эльбрус", связанные с вычислением выражений. В большинстве языков программирования арифметические операции полиморфны. Поэтому при трансляции указанного выражения на традиционной ЭВМ потребовалась бы генерация дополнительных команд преобразования значения целой переменной x в форму с плавающей точкой. В системе "Эльбрус" это преобразование выполняется автоматически, так как аппаратура во время исполнения распознает типы операндов по их тегам. Если значение x или y не определено, то одним из операндов арифметической операции *умн* будет пустой объект, что приводит к прерыванию "неверный операнд".

М а с с и в ы. В языках программирования выполняются следующие основные операции над массивами; описание; использование имени массива (например, передача его в качестве параметра); обращение к элементу; вырезка; присваивание. Все они в системе "Эльбрус" имеют аппаратную поддержку. Механизм динамического распределения памяти для массивов описан в В. 1.1.

Описание массива

var *a* : **array** [0 . . 999] **of real**

транслируется в команды формирования или загрузки *заявки на память* – слова со специальным тегом, на который реагирует аппаратура (его числовое значение равно 47). Заявка содержит следующие основные поля:

поле *форматзп* = [22 : 3], % формат массива
длиназп = [19 : 20], % длина массива
глобзп = [60]; % 0 – локальный, 1 – глобальный.

Описанию переменной-массива *a* будут соответствовать команды:

унзг (47,. (*форматзп*: 6, *длиназп* : 1000)); *пус* (0).

Заявка в системе "Эльбрус" играет роль задержанного генератора массива.

Использование имени массива. При первом использовании имени массива *a*, например *ввод* (*a*), при считывании заявки в область выражений командой *вел* или *лвел* происходит прерывание, вызов ОС и отведение математической памяти для массива *a*. Операционная система после отведения памяти заменяет в области локальных данных заявку на дескриптор массива. При последующих обращениях к массиву *a* в область выражений будет загружаться дескриптор массива. С точки зрения программиста (по внешнему эффекту) и разработчика транслятора (по генерируемой команде), первое использование *a* ничем не отличается от остальных.

Поэлементная обработка массива. Обращению к элементу массива $a[i]$ соответствуют два варианта машинной команды: *инд* – вычисление адреса элемента массива (индексация); *индсч* (индексация со считыванием) – считывание значения элемента. Если (*ур, смещ*) – адресная пара переменной-массива *a*, то выражение $a[0]+1$ транслируется в последовательность команд:

вел (*ур, смещ*); *зг0*; *индсч*; *зг1*; *сл*,

а оператор $a[0] := 1$ – в последовательность команд

вел (*ур, смещ*); *зг0*; *инд*; *зг1*; *з64и*,

где *з64и* – команда записи значения не длиннее 64 разрядов, порядок операндов которой обратный по отношению к команде *з64*.

Команда *з64и*, в отличие от *з64*, используется при генерации кода присваивания компонентным переменным, так как при однопросмотровой трансляции код вычисления адреса переменной (левой части) предшествует коду вычисления выражения (правой части). В командах *инд* и *индсч* контролируется соответствие индекса *i* размеру массива *a*: если $i < 0$ или $i >=$ **длина** *a*, происходит прерывание "граница массива".

Вырезка (*подмассив*). Для выделения подмассива $a[i : k]$ (в терминах языка Эль-76) используются две команды: *впн* (взять подмассив с начала) и *впк* (взять подмассив с конца). Смысл их ясен из следующего примера. Конструкция $a[2 : 5]$ транслируется в последовательность команд:

вел (*ур, смещ*); *зг8* (2); *впн*; *зг8* (5); *впк*.

Пересылка массивов. Если *а* и *в* — одномерные массивы одинаковой структуры, то оператор *а*<:= **ф64** транслируется в код:

вел ($ур_a$, $смещ_a$); *вел* ($ур_b$, $смещ_b$); *зг1*; *изнак*; *мпс*,

где ($ур_a$, $смещ_a$) — адресная пара *а* (аналогично — *в*); *изнак* — команда изменения знака; *мпс* — команда безусловной пересылки массивов. Команда *мпс* имеет три операнда (дескриптор получателя, источника и количество элементов) и выполняет пересылку (поэлементное присваивание) векторов формата **ф64**. Отрицательное количество означает пересылку всего вектора.

Массивы с вычисляемыми границами и многомерные массивы реализуются с помощью паспортов (п. 1.7). Аппаратную поддержку имеют операции обращения к элементу массива размерностей 1, 2 и 3. Аппаратные операции выделения подмассива и пересылки над многомерными массивами не определены.

Процедуры. Основные компоненты процедурного механизма и его семантика во всех языках практически одинаковы. Давно сложился и классический метод реализации процедур с помощью стека [58]. В системе "Эльбрус" понятие процедуры положено в основу организации программы, а метод реализации процедур, близкий к методу [58], "вложен" в аппаратуру. Следуя принятой схеме, рассмотрим основные элементы процедурного механизма языков программирования и соответствующие им машинные понятия и операции.

Описание процедуры. В процедурных языках программирования программа представляется в виде дерева вложенных процедур. Любой процедуре доступны ее локальные описания и локальные данные всех статически объемлющих процедур. В соответствии с этим любая программа в системе "Эльбрус" также состоит из процедур. Объектный код программы разбит на *сегменты*, каждый из которых логически соответствует одной процедуре (хотя возможно и объединение нескольких процедур в один сегмент с целью оптимизации загрузки их кода во время исполнения). Процедура характеризуется *номером сегмента* объектного кода, *номером начальной команды* внутри сегмента, а также *контекстом* — математическим адресом области локальных данных непосредственно объемлющей процедуры, в которой она описана, и *уровнем вложенности*. Таким образом, для ссылки на процедуру и ее аппаратного запуска используются следующие атрибуты: *уровень, номер сегмента, номер команды* и *контекст*. Они объединяются в *метку процедуры* — слово с тегом **мпроц**. Можно было бы считать, что описанию процедуры соответствует команда формирования ее метки: *фмб* (*ур*, *нсег*), где *фмб* означает "формирование метки со сменой базы" (номер сегмента); *ур* — уровень процедуры; *нсег* — номер сегмента; номер команды полагается равным нулю, а контекст — адресу области локальных данных уровня *ур*-1 (из базового регистра). Однако в системе принята следующая оптимизация: при описании процедуры метка не формируется, а для вызова процедуры используется команда "открытый вход" (она описана ниже), которая определяет контекст вызываемой процедуры не по метке, а по содержимому базового регистра.

Формирование метки выполняется только в том случае, если при исполнении необходимо явное представление процедурного объекта, на-

пример при передаче процедуры в качестве параметра или при присваивании процедурного объекта переменной процедурного типа.

Вызов процедуры. При вызове процедуры должны быть выполнены следующие действия:

– установка контекста идентификаторов, доступного процедуре;

– передача фактических параметров;

– передача управления объектному коду процедуры.

Все эти действия в системе "Эльбрус" выполняются командой входа в процедуру. Если процедура p описана в текущем контексте:

program *пример*; ...
procedure $p(a1 : t1; \ldots ; a_n : tn)$;
  begin ... **end** (* p *);
... $p(x1, \ldots, xn)$ ...
end. (* *пример* *)

то ее вызов реализуется по следующей схеме:

p $(x1, \ldots, xn)$
омс; $\langle x1 \rangle; \ldots ; \langle xn \rangle$; *овход 0* (*ур, нсег*)

Каждому синтаксическому элементу вызова соответствует определенная машинная команда. Команда *омс* – команда *открытой маркировки стека* (открывающая скобка вызова процедуры) резервирует два слова в области выражений и частично формирует в них связующую информацию (МКС и УСВ). $\langle x1 \rangle, \ldots, \langle xn \rangle$–команды вычисления фактических параметров и загрузки их в область выражений. Команда *овход0* – *открытый вход в процедуру* (закрывающая скобка). Уровень процедуры *ур* и номер ее сегмента *нсег* определяются компилятором и указываются в команде. В качестве контекста берется адрес области локальных данных уровня *ур*-1 из дисплей-регистра, т.е. адрес области локальных данных процедуры, текстуально объемлющей p.

Перед входом в процедуру завершается формирование связующей информации области локальных данных. Первое слово (*маркер стека* – МКС) содержит *контекст* вызываемой процедуры и *размер* области локальных данных. Список маркеров стека, начинающийся с области данных текущей процедуры и образуемый адресами контекста, называется *статической цепочкой.* Ее прообраз в тексте программы – цепочка статически вложенных процедур. Во втором слове (*управляющем слове возврата* – УСВ) хранятся координаты точки возврата из процедуры – ссылка на объектный код места возврата и относительный адрес соответствующей области локальных данных по стеку. Список областей данных в порядке, обратном вызову процедур, называется *динамической цепочкой.* Установка контекста при входе в процедуру выполняется аппаратно путем просмотра статической цепочки. Последним элементом статической цепочки считается дескриптор справочника сегментов кода (ССК), уровень которого условно считается равным –1 и который также может заменяться при смене контекста. Таким способом реализовано аппаратное переключение с одной исполняемой программы на другую.

После входа в процедуру p в дисплей-регистрах уровней 0, ..., *ур*-1 находятся дескрипторы областей данных глобального контекста, уровня *ур* –

дескриптор области локальных данных процедуры p (в начале эта область состоит из двух слов – МКС и УСВ). Значения параметров $x1, \dots, xn$ находятся в области выражений. Как правило, первой командой процедуры p является команда *пус* (0), которая присоединяет параметры к области локальных данных.

Передача параметров. Все основные способы передачи параметров, принятые в языках программирования, в системе "Эльбрус" реализованы аппаратно.

1. *Параметры-значения*:

procedure $p(x : t)$; **begin** ... $x + 1$ ... **end**

Передача параметра реализуется вычислением выражения, значение которого становится начальным значением формального параметра x и используется, как и значение переменной, командами *вел* или *лвел*.

2. *Параметры-переменные*:

procedure q (**var** $v : t$); **begin** ... $v := e$ ... **end**

Передача параметра реализуется вычислением *адреса* переменной (*имени* – для простой переменной, *косвенного слова* – для компонентной переменной, соответственно, командой *зга* и *инд*). Оператор присваивания в теле процедуры q транслируется в команды:

зга (*ур, смещ*); *савел*; ⟨вычисление *e*⟩; *з64и*,

где (*ур, смещ*) – адресная пара v; *савел* – команда считывания адреса величины ("разыменования" переменной v вплоть до последнего адреса, указывающего на фактическую переменную). Использование значения v реализуется командой *вел* (*лвел*), которая в данном случае осуществляет просмотр цепочки адресов вплоть до значения фактической переменной.

3. *Параметры-процедуры и функции*:

procedure *r* (**procedure** *p* (*y* : *integer*));
  begin ... *p*(1) ... **and** (* *r* *);
procedure *s* (*a* : *integer*); **begin** ... **end** (* *s* *); ...
  begin ... *r*(*s*) ... **end**

Передача параметра-процедуры S реализуется командой формирования ее метки *фмб*, выполнение которой описано выше. Вызову параметра-процедуры p в процедуре r соответствует последовательность команд:

p	(	1	);
вел (*ур, смещ*);	*мс*;	*зг*1;	*вход*0.

Здесь команда *вел* считывает метку процедуры p; команда *мс* (маркировка стека) – подготовительная команда для входа, аналогичная *омс* (образует на месте метки МКС и резервирует в стеке одно слово для УСВ); *вход*0 – команда входа в процедуру по метке. Действия команд *мс* и *вход*0 аналогичны действиям команд *омс* и *овход*0, но информация о контексте процедуры берется не из дисплей-регистра, а из метки процедуры p.

4. *Передача параметра по наименованию* (в языке Эль-76 – **прог** или **имя прог**, см. п. 1.16.2):

процедура p = **проц** (*a*) (... *a* ...);
p (**прог** $x + y$)

Классическим способом реализации этого вида параметров [58] является передача подпрограммы вычисления значения (адреса) параметра. В системе "Эльбрус" этот способ имеет аппаратную поддержку. Выделен специальный класс процедур – *технические процедуры*, отличающиеся от обычных тегом метки. Техническая процедура запускается автоматически без параметров при считывании ее технической метки. Передача фактического параметра реализуется "запроцедуриванием" вычисления $x + y$ и преобразованием метки процедуры в техническую командой *техмет* (в системе нет команды, которая сразу формирует техническую метку). Использование формального параметра a реализуется, как и при предыдущих способах передачи, командой *вел* (*лвел*), которая автоматически "распроцедуривает" параметр. Таким образом, команды объектного кода, реализующие использование формального параметра, не зависят от способа его передачи, что отражено в базовом языке Эль-76, в котором способ передачи специфицируется при вызове.

Процедуры-функции. Вызов функции, в отличие от процедуры, реализуется командой *овход*1 (*вход*1), которая передает результат из вершины области выражений в область выражений вызывающей процедуры. Если результат функции f задается оператором присваивания $f := e$ (как в языке Паскаль и Алгол-60), то, как и при реализации на традиционных ЭВМ, для результата функции формируется внутренняя локальная переменная (*ячейка-результат*).

Выход из процедуры в языках программирования выполняется конструкцией вида: **return** (или **return** (x)), где x – результат функции, либо после выполнения текстуально последнего оператора процедуры. В обоих случаях в системе "Эльбрус" он реализуется командой *выход*. Если процедура является функцией, то предварительно в область выражений должен быть загружен результат (значение операнда x конструкции **return** или ячейки-результата f). Команда *выход* восстанавливает контекст вызывающей процедуры по ее статической цепочке, хранящейся в стеке, и обеспечивает продолжение ее выполнения. Если процедура была вызвана как функция (командой *вход*1), то содержимое вершины стека переносится в область выражений вызывающей процедуры в качестве результата.

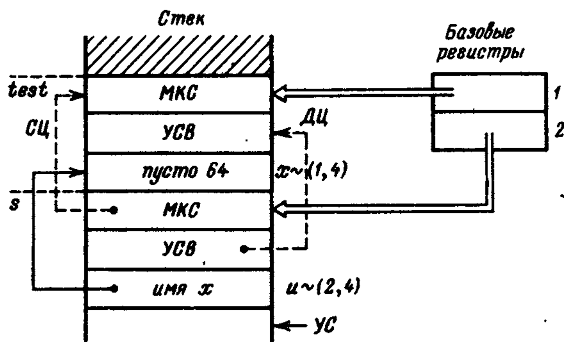
Пример. Для иллюстрации работы аппаратного процедурного механизма МВК "Эльбрус" рассмотрим простой модельный пример на языке Паскаль:

```
program test;
    var x : real;
    procedure q (function f (x : real) : real);
          var y : real; (* 2 *)
          procedure r;
                var z : real; (* 3 *)
                begin z := 1. 0;
                      y := f (z)
                end (* r *);
       begin (* q *)
             x := 2. 0;
             r
       end (* q *);
```

```
procedure s (var u : real); (* 1 *)
        function g (t : real) : real;  (* 4 *)
        begin g := u + 1
        end (* g *);
   begin (* s * )
        u := 3.0;
        q(g)
   end (* s *);
begin
     s(x)
end. (* test *)
```

Приведем объектный код и адресные пары переменных и параметров программы *test* и составляющих ее процедур *q*, *r*, *s* и функции *g*. Номера программных сегментов: *test* – 10, *q* – 11, *r* – 12, *s* – 13, *g* – 14.

1) Перед исполнением тела s



2) Перед исполнением тела q

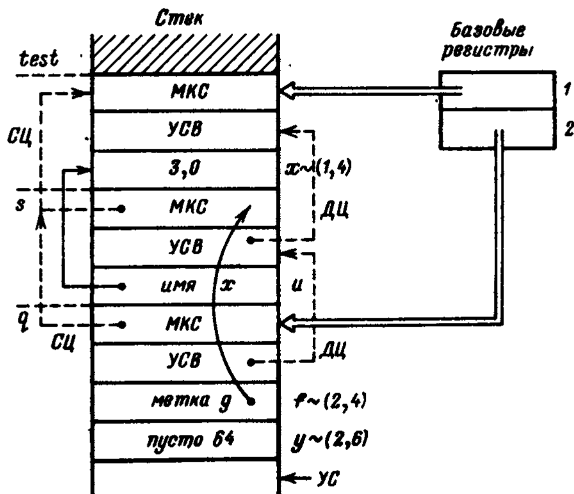


Рис. 5

Рис. 5 (окончание)

П р о г р а м м а *test*. Уровень: 1. Номер сегмента: 10. Адресные пары: x = (1, 4). Объектный код:

пус (1); *омс*; *зга*(1,4); *овход*0 (2,13); *выход*.

Процедура q. Уровень: 2. Номер сегмента: 11. Адресные пары: f = = (2,4), y = (2,6). Объектный код:

пус(1); *унзг*(*вещ*64, 2.0); *зга*(1,4); *з64*; *омс*; *овход*0 (3,12); *выход*.

Процедура r. Уровень: 3. Номер сегмента: 12. Адресные пары: z = (3,4). Объектный код:

пус(0); *унзг*(*вещ*64, 1.0); *зга*(3,4); *з*64; *вел*(2,4); *мс*; *вел*(3,4); *вход*1; *зга*(2,6); *з*64; *выход*.

Процедура s. Уровень: 2. Номер сгемента: 13. Адресные параметры: u = (2,4). Объектный код:

пус(0); *зга*(2,4); *савел*; *унзг*(*вещ*64, 3.0); *з*64*u*; *омс*; *фмб*(3,14); *овход*0(2,11); *выход*.

Функция g. Уровень: 3. Номер сегмента: 14. Адресные пары: t = (3,4); g = (3,6) (ячейка-результат). Объектный код:

пус(1); *вел*(2,4); *зг*1; *сл*; *выход*.

Последовательность состояний стека в моменты 1 (перед исполнением тела процедуры s), 2(перед исполнением тела q), 3 (перед исполнением тела r) и 4 (перед исполнением тела f) показана на рис. 5. В объектном коде программы *test* для простоты опущено подключение модуля динамической поддержки системы Паскаль-Эльбрус. Двойными стрелками на рис.5 показаны ссылки из базовых регистров на области локальных данных, пунктирными стрелками – элементы статической цепочки (СЦ) и динамической цепочки (ДЦ). Адресными парами помечены доступные в данный момент идентификаторы.

Управляющие конструкции. К средствам управления программой в языках программирования относятся: метки и операторы перехода (исторически первые управляющие конструкции); условные предложения; циклы; ситуации и структурные предложения. Аппаратные средства управления программой в МВК "Эльбрус" предназначены для адекватного отображения управляющих конструкций языков программирования в объектный код. Они включают: традиционный набор команд управления (безусловные и условные переходы; команды организации циклов); аппаратный тип данных "метка перехода" и команду динамического перехода по метке (с возможной сменой контекста); команду структурного перехода по динамической ситуации и ее программное "продолжение" в виде процедуры операционной системы.

М е т к и и п е р е х о д ы. Метка в языке программирования служит для обозначения точки в программе. С понятием метки связаны два элемента языка:

– *определяющее вхождение метки M*;

– *оператор перехода* **goto** *M*.

В некоторых языках (например, Эль-76 и Паскаль) обязателен и третий элемент – предварительное описание метки в блоке, к которому относится ее определяющее вхождение.

Будем различать *локальные* и *глобальные переходы* по метке. Переход называется локальным, если оператор перехода и определяющее вхожде-

ние метки находятся в одном блоке, в противном случае – глобальным. Глобальный переход по метке связан с изменением контекста идентификаторов, поэтому для глобальных переходов информация о точке в программе должна включать ее *контекст*.

В системе "Эльбрус" текст некоторого блока (процедуры) транслируется, как правило, в один программный сегмент. Точке в программе внутри блока при этом соответствует номер начального байта команды в сегменте. Для реализации локальных переходов служит команда *бпн* α, где *бпн* – безусловный переход непосредственный; α – положительное или отрицательное смещение номера байта команды, на которую выполняется переход, относительно номера байта данной команды *бпн*. Относительная адресация команд при локальных переходах облегчает оптимизацию программ, так как позволяет перемещать код внутри сегмента. В примерах данной главы номера байтов команд и их относительные адреса при переходах будут обозначаться буквами греческого алфавита: $\alpha, \beta, \ldots$

При глобальном переходе точек в программе соответствует информация, аналогичная информации о процедуре для ее запуска: *уровень и номер сегмента* процедуры, в которой описана метка; *номер байта*, соответствующий определяющему вхождению метки, и *контекст* (адрес области локальных данных процедуры). Для хранения этой информации служит специальный тип данных – *метка перехода* (тег – мпход). Метка перехода создается командой *формирования метки перехода*:

фмбп (*ур*, *нсег*, *нк*)

(переход в другой программный сегмент) или:

фмп (*ур*, *нк*)

(переход в тот же программный сегмент). Здесь: *ур* – уровень, *нсег* – номер сегмента, *нк* – номер команды. Контекст берется из базового-регистра с номером *ур*, в отличие от метки процедуры, которая ссылается на область данных *предыдущего* уровня. Переход по метке выполняется командой *бпд* (безусловный переход динамический), которая берет метку перехода из стека. При глобальном переходе уничтожается часть стека до границы области локальных данных процедуры, куда происходит переход, и устанавливается контекст этой процедуры.

П р и м е р.

```
program p (output);
label 1;
procedure q;
  begin ... goto 1 ... end (* q *); ...
1 : writeln ('конец')
end. (* p *)
```

Оператор перехода **goto** 1 из процедуры q в основную программу p транслируется в команды:

фмбп (1, 10, α); *бпд*,

где α – номер байта в сегменте номер 10, соответствующий определяющему вхождению метки 1.

Условные предложения реализуются командами условного перехода: *у0н* α и *у1н* α – условный переход по нулю (единице) непосредственный. В стеке – значение условия; команда *у0н* вычеркивает условие и выполняет переход на команду α, если младший бит условия равен нулю; команда *у1н* выполняет аналогичные действия, если младший бит условия равен 1. Схема реализации условного предложения:

if B **then** $s1$ **else** $s2$
$\langle B\rangle$; *у0н* $\alpha 2$; $\langle s1\rangle$; *бпн* α; $\alpha 2 : \langle s2\rangle$; $\alpha : \ldots$

Циклы вида **while** B **do**S и **repeat** S **until** B, как и условные предложения, транслируются в команды условного и безусловного перехода по следующим схемам:

while B **do** S
α: $\langle B\rangle$; *у0н* β; $\langle S\rangle$; *бпн* α; β :
repeat S **until** B
α : $\langle S\rangle$; $\langle B\rangle$; *у0н* α;

Для реализации циклов вида **for** $i := m$ **to** n **do** S и **for** $i := m$ **downto** n **do** S предназначены специальные команды. Поскольку для языков программирования типично использование нескольких вложенных циклов типа **for** (как правило, до трех) для обработки многомерных массивов, в аппаратуре выделено четыре специализированных регистра параметров цикла *рц0*, *рц1*, *рц2* и *рц3*. Регистр с меньшим номером соответствует параметру цикла с большей глубиной вложенности; в нулевом регистре хранится параметр самого внутреннего цикла. Приведем схему реализации цикла:

for $i := m$ **to** n **do**
  α : $\langle m\rangle$; $\langle n\rangle$; *нцс*
begin . . . i . . . **end**
счт пц 0; . . . *кц* α

Команда *нцс* (начало цикла со сложением) формирует нулевой регистр параметра цикла с начальным значением m, шагом 1 и конечным значением n. Для освобождения нулевого регистра цикла выполняется групповое присваивание; *рц*3 := *рц*2; *рц*2 := *рц*1; *рц*1 := *рц*0. Для цикла **downto** используется команда *нцв* (начало цикла с вычитанием), задающая шаг – 1. Команда *счтпц* κ ($\kappa = 0, 1, 2, 3$) считывает значение κ-го регистра параметра цикла в стек; она соответствует использованию параметра цикла i в теле цикла. Команда *кц* α (конец цикла) модифицирует параметр цикла, выполняет проверку на его окончание и в случае продолжения цикла осуществляет переход к команде α (в начало цикла). По окончании цикла команда *кц* возвращает регистры цикла в исходное состояние: *рц*0 := *рц*1; *рц*1 := *рц*2; *рц*2 := *рц*3.

Если из тела цикла выполняется переход (например, **goto** 1), прекращающий его выполнение, то в объектном коде команде перехода должна предшествовать команда *вц* (выход из цикла), которая восстанавливает исходное состояние регистров цикла.

С и т у а ц и и . Наиболее часто используются *статические ситуации*, определяемые в структурном предложении (п. 1.15.1). Их применение в программе не приводит к ухудшению ее эффективности, поскольку они

реализуются обычными командами переходов. Например:

до *конец*
(... *конец*! ...)
бпн α; ... *бпн* β;
при *конец*: S
α: ⟨S⟩
всесит
β: ...

Более сложной является реализация *динамических ситуаций* (п. 1.15.3). Понятию динамической ситуации соответствует системное понятие "*имя ситуации*" – дескриптор, ссылающийся на место определения динамической ситуации S и используемый для поиска ее реакции. Для обработки динамической ситуации S структурным предложением в реализации организуется *ловушка* – два слова в области локальных данных, содержащие имя ситуации S и метку перехода на реакцию. Область локальных данных процедуры, содержащей это структурное предложение, помечается специальным признаком прерывания с помощью процедуры ОС *пометкаусв* (a), где a – адрес области локальных данных. Структурный переход $s!(x_1, \ldots, x_n)$ реализуется командами:

⟨x_1⟩; , ... , ; ⟨x_n⟩; *зг*8 (n); *вел* (*ур*$_S$, *смещ*$_S$); *бпд* ,

где n – число параметров структурного перехода; (*ур*$_S$, *смещ*$_S$) – адресная пара идентификатора S; *бпд* – команда динамического безусловного перехода. Команда *вел* считывает в стек имя ситуации S. Команда *бпд*, если ее операндом является не метка перехода, аппаратно просматривает динамическую цепочку до ближайшей помеченной области локальных данных. При ее обнаружении происходит прерывание и вызывается процедура ОС, которая проверяет, есть ли в найденной области ловушка на ситуацию S. Если ловушка найдена, то уничтожается часть стека до границы найденной области локальных данных, параметры $x_1, \ldots, x_n$ переписываются в область выражений и выполняется переход по метке на реакцию. В противном случае вновь продолжает работать команда *бпд* (если ловушка не будет обнаружена в текущем процессе, он уничтожается, и распространение ситуации S продолжается в процессе-хозяине; в конечном итоге отсутствие ловушки приводит к перерыванию "неизвестная системе ситуация").

Пример. Схема реализации динамической ситуации S.

конст S = *ситуация* ( );
пус (1); *зга* (*ур*$_S$, *смещ*$_S$); *ир* (*деск*, *ф*64, 1); *зга* (*ур*$_S$, *смещ*$_S$); з...
...
до * S
омс; *зга* (*ур*$_S$, 0); *овхдс*0 (*пометкаусв*);
вел (*ур*$_S$, *смещ*$_S$); *фмп* (*ур*, α); *пус* (0);
(... p( ) ...)
бпн β
при S : R
α : ⟨R⟩
всесит;
β; ...
процедура p = **проц** ( ) (... S! ...);
*зг*0; *вел* (*ур*$_S$, *смещ*$_S$); *бпд*

Описанию идентификатора S соответствуют команды записи в область локальных данных по адресной паре ($ур_S$, $смещ_S$) имени ситуации S – дескриптора, ссылающегося на это же слово. Дескриптор формируется из имени слова S командой преобразования адресной информации *ир* (изменить размер). Структурное предложение для обработки ситуации S находится в процедуре уровня *ур*. Его заголовок (до * S) транслируется в команды вызова системной процедуры *пометкаусв* (*овхдс*0 – открытый вход в системную процедуру) и формирования ловушки. Реакция и конец структурного предложения реализованы с помощью команд перехода. Структурный переход *s*! в процедуре *p* транслируется по описанной выше схеме (число параметров *n* равно нулю).

Модули. Процедурный механизм системы "Эльбрус" содержит не только аппаратную реализацию традиционных средств работы с процедурами, принятых в алголоподобных языках, но и реализацию процедур с *собственным контекстом*. При исполнении программы на языке Алгол-60 или Паскаль локальные данные всех процедур размещаются в стеке. Однако в некоторых языках программирования требуется организация процедур, которые могут обращаться к элементам контекста идентификаторов, находящимся вне стека в различных областях памяти. Наиболее известный пример – подпрограмма в языке Фортран, которой доступны общие данные, описанные в одном или нескольких COMMON-блоках. Более современный пример – языки модульного программирования (Эль-76, Клу, Модула-2 и другие), в которых группа процедур, работающих с общими данными (последние можно рассматривать как конкретное представление некоторого объекта), объединяется в *модуль* (пакет), состоящий из *интерфейса* (внешних описаний процедур) и *реализации* (описаний тел процедур и их общих данных). Процедура интерфейса модуля должна работать в контексте его интерфейса и реализации, но сама реализация должна быть скрыта от пользователя модуля.

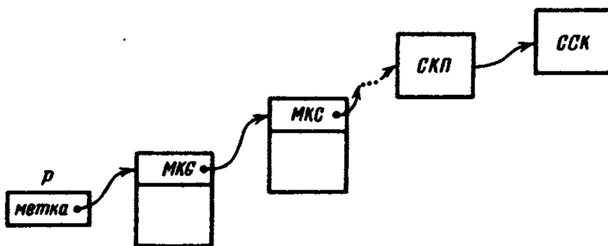


Рис. 6

Возможности процедурного механизма МВК "Эльбрус" позволяют единым образом решить все эти проблемы. Для формирования метки процедуры *p*, работающей в собственном контексте, служит команда *уфм* (универсальное формирование метки). Ее операндами в стеке являются: заготовка метки, содержащая только ссылку на код (без информации о контексте), и адрес (например, имя или дескриптор). Команда *уфм* формирует из заготовки метку процедуры *p*, контекст которой задается указанным адресом.

Возможны два варианта использования процедуры с собственным контекстом, которым соответствуют различные способы установки контекста на базовые регистры при входе в процедуру.

1. Если процедура *p* принадлежит использующей программе, а ее собственный контекст имеет иерархическую структуру, то последовательность областей собственного контекста можно задать цепочкой маркеров стека (см. рис. 6). В конце этой контекстной цепочки всегда находятся две системные области, характеризующие данную программу: *область собственного контекста программы* (СКП), создаваемая при ее открытии и содержащая ссылки на постоянные для программы объекты, и *справочник сегментов кода* (ССК), содержащий дескрипторы сегментов кода программы. Установка контекста при входе в процедуру *p* в этом случае выполняется так же, как и при обычном входе.

2. Если процедура *p* является независимо откомпилированной либо порядок областей в контекстной цепочке может меняться для разных процедур, то следует применить другой способ (см. рис. 7). Адрес из метки *p* должен указывать на дескриптор вектора формата **ф64**, содержащего дескрипторы областей произвольного контекста в порядке убывания их предполагаемых уровней (этот вектор называется *пачкой дескрипторов*). Последнее слово пачки дескрипторов должно содержать адрес продолжения контекстной цепочки, который, как правило, указывает на маркер стека. При входе в процедуру *p* на базовые регистры уровней $yp-1, \ldots$ $\ldots, yp-k+1$ загружаются дескрипторы $n[0], \ldots, n[k-2]$ (где *yp* – уровень процедуры *p*; *k* – длина пачки; *n* –дескриптор пачки). Слово $n[k-1]$ управляет дальнейшей установкой дисплей-регистров; адрес может указывать на МКС или вновь на пачку и т.д.

В качестве примера использования первого способа организации собственного контекста рассмотрим реализацию модульных объектов языка Эль-76 (п. 1.31). Пусть в программе имеются следующие описания интер-

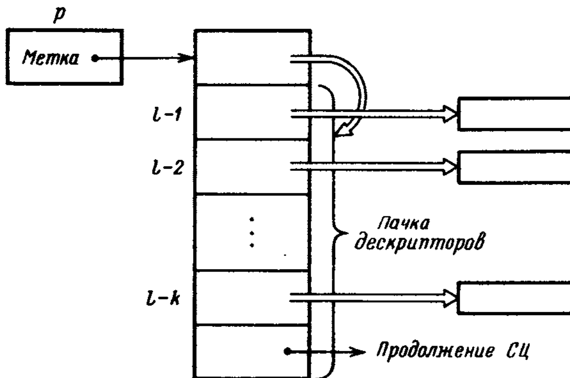


Рис. 7

фейса и реализации модуля:

```
тип инт = интерфейс
                ф64 p, q, r
           конинт;
конст реал = реал инт
                начало
                   ф64 a, b;
                   процедура pp = проц(x) (. . . a . . . b . . . x . . . q . . . r);
                   . . .
                   p := pp
                конец;
% – – – использование:
перем m # инт := (реал); . . .
m.p (1)
```

Состояние стека, памяти и дисплей-регистров после вызова процедуры *p* из модульного объекта *m* показано на рис. 8. Области памяти для контекста интерфейса и реализации, их контекстные связи через маркеры стека и метка процедуры *p* с собственным контекстом формируются процедурой *ген*. Процедура *p* имеет уровень 3; ее контекст задается адресом области контекста реализации. При исполнении процедуры *p* ей доступны локальные данные и параметры (уровень 3), элементы реализации (уровень 2) и интерфейса (уровень 1).

Пример второго способа организации собственного контекста связан с независимо откомпилированной процедурой. Если процедуре *p* на языке Эль-76 должны быть доступны области контекста *c*1, *c*2, *c*3, то ее следует оттранслировать как отдельную программу в специальном режиме (с установкой уровня):

```
¤ уст уровень=4
p = проц (x)
начало . . .
конец % //p – внешнее имя файла объектного кода.
```

Для вызова процедуры *p* в указанном контексте следует предварительно сформировать соответствующую метку (см. рис. 7). Эти действия выполняет системная процедура *ген*:

```
ф64 p := ген (//p, c1, c2, c3);
p (1)
```

Таким образом, архитектура системы "Эльбрус" обеспечивает поддержку реализации всех классов конструкций процедурных языков программирования, включая как более традиционные средства (процедуры, операторы), так и современные возможности – ситуации и модули.

Общая схема функционирования компилятора в системе "Эльбрус" изображена на рис. 9. Прямоугольными рамками выделены компоненты программного обеспечения, овальными – системные структуры данных и файлы. Простыми стрелками показаны входные и выходные данные компилятора, полужирными – обращения к программным компонентам, двойными – обращения к системным структурам

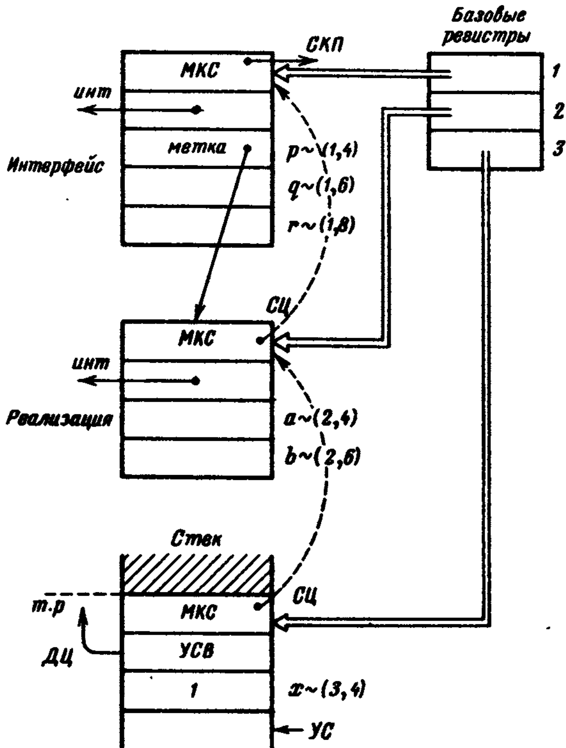


Рис. 8

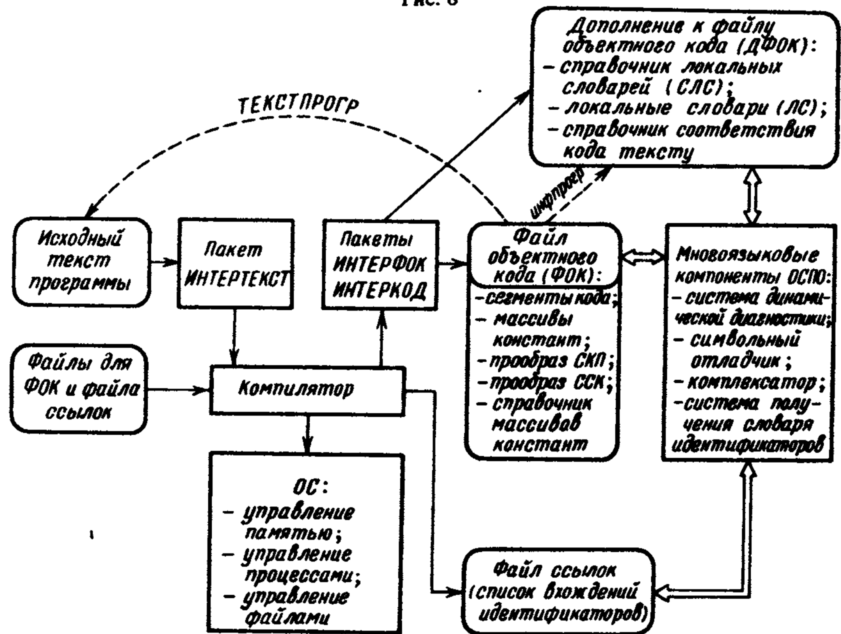


Рис. 9

данных, пунктирными – стандартные ссылки между файлами, устанавливаемые компилятором. Компилятор получает на вход исходный текст программы, файлы для генерации объектного кода и словаря ссылок (в режиме "словарь"). Выходными структурами данных являются: расширенный файл объектного кода (РФОК), состоящий из файла объектного кода (ФОК) – кодового представления программы и дополнения к файлу объектного кода (ДФОК) – символьно-табличного представления программы, – и словарь (файл) ссылок (список всех вхождений идентификаторов), который генерируется только в специальном режиме. Компоненты этих системных файлов показаны на рисунке; их смысл разъяснен в В.1.4. РФОК используется многоязыковыми компонентами ОСПО для сборки, динамической диагностики, отладки и документирования программы.

При обработке исходного текста компилятор использует технологический пакет ИНТЕРТЕКСТ, при генерации РФОК – технологический пакет ИНТЕРФОК (генерация стандартной структуры РФОК) и технологический пакет ИНТЕРКОД (генерация объектного кода конкретной модели МВК "Эльбрус").

Во время исполнения программы, полученной в результате компиляции, используются следующие программные компоненты:

– система динамической поддержки данной системы программирования;

– операционная система и система файлов;

– технологический пакет ИНТЕРПАМ – для работы с памятью задачи, при сборке мусора и других видах динамического анализа программы или ее данных;

– система динамической диагностики – при динамических ошибках;

– символьный отладчик – при отладке программы в диалоге.

6.1.2. Динамические языки. Этот класс языков содержит многие распространенные языки программирования, используемые в задачах обработки символьной информации и в прикладных системах искусственного интеллекта (Лисп, АБВ, Снобол-4, Плэнер, Пролог, Рефал). Их характерными свойствами являются:

– динамический контроль типов;

– сложные механизмы именования (ассоциативные списки, динамическая идентификация имен);

– динамические структуры данных (строки, таблицы, списки) и сложные по семантике операции над ними (сравнение с образцом, синтаксическое отождествление и т.п.), выражающие различные формы унификации;

– нетрадиционные средства управления, связанные с анализом сложных структур данных (возвраты; переходы при успехе и неудаче; прерывания и прекращения);

– расширяемость программы во время ее исполнения.

Все эти свойства при реализации на традиционных ЭВМ неизбежно приводят к необходимости интерпретации и создают устойчивое представление программистов-пользователей о неэффективности использования этих языков. Однако возможности системы "Эльбрус" изменяют эти сложившиеся представления. Архитектура системы "Эльбрус" не была специаль-

но ориентирована на данный класс языков. Для такой ориентации (как видно хотя бы из беглого сравнения с известной ЭВМ Symbol [48], ориентированной на обработку символьной информации), необходим совершенно иной подход к архитектуре – встроенная списочная память, аппаратные операции конкатенации и расширения списков. Однако элементы динамизма, заложенные в аппаратуру системы "Эльбрус", дают возможность эффективно реализовать многие из перечисленных механизмов динамических языков, что подтверждается реализацией для МВК "Эльбрус" языков Лисп, АБВ, Снобол-4 и Рефал. Для некоторых из них, например для языка АБВ (пп. 4.1, 6.3.3), становится возможной разработка компилятора, генерирующего эффективный код; для других, например для Снобола-4 (п. 4.2), архитектура МВК "Эльбрус" способствует повышению эффективности интерпретатора.

Проанализируем более подробно свойства архитектуры системы "Эльбрус", облегчающие эффективную реализацию динамических языков. Изложение этих свойств ведется в основном в терминах языка Эль-76 с указанием в необходимых случаях соответствующих машинных команд.

Динамический контроль типов. Основой для представления информации о типах данных во время счета и для реализации динамического контроля типов в интерпретационных языках являются теги. Каждый простой объект имеет свой тег, который контролируется аппаратурой при выполнении машинных команд и может быть выделен операцией тип *x* (соответствующая машинная команда – *счтег*). С помощью тегов может быть представлена информация о стандартных типах данных: числах, логических значениях, символах, адресах, процедурах, неопределенных значениях. Контроль типов для этих объектов ведется аппаратно.

Для представления информации о специфических типах данных динамического языка (например, STRING и PATTERN и Сноболе-4, СТРОКА и ЦЕЛ в АБВ) могут быть использованы *теги*, не занятые для кодировки других стандартных типов языка. Таким образом, значение объекта языкового типа может быть представлено ссылкой (индексом) или несколькими ссылками, упакованными в слово и снабженными тегом, который в данной реализации выделен для кодировки рассматриваемого языкового типа.

Для повышения эффективности интерпретационных проверок, выполняемых в динамическом языке для каждого вхождения идентификатора (например, проверки на неопределенное значение, со специфической реакцией на ошибку), удобно использовать встроенный интерпретационный механизм системы "Эльбрус" – *технические процедуры*. Неопределенное значение объекта может быть представлено технической меткой процедуры интерпретации соответствующего прерывания. При использовании (чтении) объекта эта процедура будет автоматически запущена, если значение объекта не определено, иначе в стек будет считано текущее значение объекта. Тем самым, обращение к объекту в динамическом языке реализуется столь же эффективно, как и в обычном процедурном языке.

Еще одна интересная возможность системы "Эльбрус" связана с *доопределением машинных операций*. Пусть, например, тип значений языкового типа СТРОКА представлен тегом, не занятым для других типов языка. Машинная команда сложения *сл* может быть доопределена для строк

как конкатенация. Для этого процедура реакции на стандартную ситуацию *неверныйоперанд* должна быть переопределена таким образом, что при сложении двух строк вызывается соответствующая интерпретирующая процедура. Такое переопределение может быть выполнено средствами технологического пакета ИНТЕРПАМ.

Динамические структуры данных интерпретационных языков программирования могут быть представлены с помощью векторов переменной длины. Для каждого языкового типа данных отводится отдельный *пул* – вектор переменной длины, в котором размещаются объекты данного типа. Объект представлен его длиной и индексом в пуле, упакованными в слово; тип объекта кодируется специально выделенным тегом. Такое представление используется, например, для объектов вида ЦЕЛ (целые переменной длины) в языке АБВ. Локализация представления динамических объектов каждого типа облегчает управление памятью: выделение, освобождение и сборку мусора. Эти идеи развиты в работах А.Н. Смертина по системе Снобол-Эльбрус [71]. Использование векторов переменной длины для представления строк в языках символьной обработки описано в п. 6.2.2.

Именование. Элементы аппарата имен в разных динамических языках существенно различаются (по крайней мере, по форме). В языках Лисп и Снобол-4 принята динамическая идентификация, в языке АБВ – *квазилокальный контекст* (исполняемые описания, механизм которых близок блочной структуре, см. пп. 4.1, 6.2.3), в языках Плэнер, Пролог и Рефал – различные формы сопоставления идентификаторов фрагментам обрабатываемого текста или применяемых правил вывода. В системе "Эльбрус" нет единого аппаратного механизма, адекватно соответствующего хотя бы одному из этих способов именования. В частности, динамический контекст языка Снобол-4 не может быть реализован непосредственно с помощью процедурного стека и базовых регистров, так как при динамической идентификации локальным переменным не может быть статически сопоставлен какой-либо определенный уровень. Однако для реализации аппарата имен можно использовать комбинации различных способов ассоциативного поиска, имеющихся в аппаратуре МВК "Эльбрус":

– просмотр косвенной цепочки адресов при выполнении команд считывания из памяти;

– автоматический запуск технической процедуры при считывании ее технической метки; техническая процедура может выполнять идентификацию или интерпретировать прерывание при ее неудачном исходе.

В п. 6.2.3 описан способ эффективной реализации аппарата имен языка АБВ с помощью указанных средств.

Управление. Для интерпретации возвратов при неудачном сопоставлении с образцом или синтаксическом отождествлении удобно использовать динамические ситуации. Перед началом сопоставления устанавливается ловушка на динамическую ситуацию *возврат*, а собственно возврат реализуется структурным переходом по этой динамической ситуации. Описанный метод применен для реализации механизма прекращения в первой версии системы программирования АБВ-Эльбрус [41] и для реализации сопоставления с образцом в системе Снобол-Эльбрус [12].

С р е д с т в а   р а с ш и р е н и я   п р о г р а м м ы. В любом динамическом языке имеются операции, выполняющие в той или иной форме динамическую трансляцию фрагмента программы и присоединение его к основной программе. Например, в языке АБВ это операция преобразование строки в действие (ПСД *S*) (см. п. 4.1). Реализация операций расширения программы в компиляторе с динамического языка может быть выполнена с использованием процедур операционной системы:

прогр (*p*) – открытие независимой программы с файлом объектного кода фрагмента программы *p*;

ген (*p*, *c*) – открытие независимо откомпилированной процедуры как модульного объекта в контексте, определяемом адресом *c*.

Первый вариант используется в случае, если при запуске расширения не требуется установки какого-либо контекста, второй вариант – если расширение должно запускаться в контексте места операции его динамической трансляции (как в языке АБВ). В качестве адреса контекста *c* передается адрес маркера стека объемлющей процедуры. В обоих случаях *p* – ссылка на независимый (временный) файл объектного кода, полученный в результате динамического обращения к компилятору и трансляции расширения.

С т р у к т у р а   и н т е р п р е т а т о р а. Основой для организации интерпретатора в системе "Эльбрус" является процедурный механизм языка Эль-76, имеющий полную аппаратную поддержку (пп. 1.16, 6.1.1). Интерпретация каждой конструкции выполняется отдельной процедурой. Наиболее удобной формой внутреннего представления программы для интерпретации является ее префиксная запись, которая позволяет интерпретировать операнды в зависимости от предварительно прочитанного кода операции; префиксная запись хранится в байтовой строке переменной длины [63]. Использование байтовой строки позволяет сэкономить память (по сравнению с представлением в виде вектора слов или полуслов): для кода операции и в большинстве случаев для ее непосредственного операнда требуется по одному байту. Переменная длина байтовой строки обеспечивает возможность расширения программы. Код операции служит индексом в векторе меток процедур интерпретации, который располагается в области локальных данных.

По таким принципам организован интерпретатор базы языка АБВ в первой версии системы АБВ-Эльбрус [41] и интерпретатор языка Снобол-4 в системе Снобол-Эльбрус [12].

6.2. Новые методы реализации

При разработке систем программирования Паскаль, Клу, АБВ, Снобол, Рефал, ДИАШАГ и Форт для МВК "Эльбрус" были тщательно изучены и использованы все элементы архитектуры и базового программного обеспечения системы "Эльбрус" – набор ее машинных понятий и операций (процедуры, стек, массивы, ситуации), средства модульного программирования, система файлов, средства управления памятью и процессами в ОС, технологические пакеты ИНТЕРФОК, ИНТЕРКОД и ИНТЕРПАМ, многоязыковые компоненты ОСПО. Опыт и результаты этих исследований описаны в настоящей книге и в работе [67], где дан обзор разработанных

методов и полная библиография. В ходе этих работ созданы новые методы трансляции, базирующиеся на максимальном использовании возможностей МВК "Эльбрус". Систематическое изложение всех этих методов и научных результатов, полученных за десятилетний период разработки систем программирования для МВК "Эльбрус" в ЛГУ, могло бы стать темой для отдельной монографии. В этом параграфе описаны лишь некоторые из них, отражающие различные элементы разработки транслятора – анализ описаний, представление данных, реализацию аппарата имен, семантический анализ, организацию статической и динамической поддержки модульных программ.

6.2.1. Анализ системы определений. Современные языки программирования (Ада, Модула-2, Клу и др.) характеризуются сравнительно простым синтаксисом и весьма сложной семантикой, включающей концепции статической типизации, абстрактных типов данных, модулей и др.). Естественным следствием усложнения семантики является более сложная структура описаний, чем в первых языках программирования (Фортран и Алгол). К описаниям переменных и процедур, которые входят в большинство языков программирования, добавились новые классы описаний, которые впервые появились в Алголе-68, а в языке Клу объединены общим понятием *тождества* (equates). Тождества – это описания констант, в правых частях которых могут быть сложные константные выражения, и описания (определения) типов, вводящие обозначения для часто используемых в программе изображений сложных типов данных. Примеры тождеств приведены в п. 3.4.3. В последовательности описаний-тождеств, как правило, разрешено использование идентификаторов до их определения: в языках Клу и Алгол-68 – без ограничений, в языке Паскаль – только использование еще не определенного идентификатора типа t в изображении типа-указателя $\uparrow t$. К этому же классу описаний-тождеств относятся также описания взаимно рекурсивных функций в языках функционального программирования и описания зависимостей между величинами вычислительной модели в языке Утопист системы ПРИЗ [72].

Поскольку завершение анализа описаний является отправной точкой последующего семантического анализа, наиболее желательна разработка однопросмотровых алгоритмов синтаксического и семантического анализа описаний, что для рассмотренного класса описаний-тождеств представляет определенную трудность. Если в описаниях-тождествах запрещена явная и неявная рекурсия, то они могут быть полностью проанализированы обычными методами (например, рекурсивным спуском) за два просмотра; если же рекурсия разрешена, то чисто просмотровые методы анализа вообще неудобны, так как анализ рекурсивных связей может потребовать обращения к любому описанию в любой момент анализа.

Предлагается метод [61, 66] анализа последовательности описаний-тождеств, основанный на однопросмотровом алгоритме синтаксического анализа методом рекурсивного спуска и построении списков использующих вхождений идентификаторов, которые появляются до определения. Метод описывается в общем виде, но для простоты изложения подробности формализации используемых понятий и реализации алгоритма опущены (они приведены в работах [63, 66]).

Системой определений назовем конечную последовательность равенств вида

$$D = \{ i_k = t_k \}_{k=1,\ldots,n},$$

где i_k – идентификаторы, t_k – выражения, содержащие использующие вхождения этих идентификаторов. Будем предполагать, что:

– синтаксис правых частей t_k допускает детерминированный $LL(1)$-разбор;

– областью определения любого из идентификаторов i_k является весь текст системы определений, т.е. идентификатор i_k может быть использован в любой из правых частей t_j, в том числе и при $j < k$, т.е. до своего определения;

– семантический анализ (вычисление семантических атрибутов) правой части определения t_k при условии, что все используемые в нем идентификаторы определены, может быть выполнен обычным способом в процессе синтаксического анализа t_k методом рекурсивного спуска.

Пусть T_k – дерево вывода правой части определения t_k; I_j – нетерминальные вершины деревьев вывода T_k, из которых выводятся использующие вхождения i_k. ***Графом зависимостей*** $G(D)$ системы D назовем орграф, полученных из прямой суммы деревьев T_k добавлением всевозможных дуг, направленных из каждой терминальной вершины, соответствующей использующему вхождению идентификатора i_k, в корень дерева вывода t_k правой части определения i_k.

Система определений D *корректна*, если:

(1) каждый используемый в выражениях t_k идентификатор имеет свое определение в системе D;

(2) $i_k \neq i_l$ при $k \neq l$ (недопустимы повторные определения);

(3) в графе $G(D)$ нет ориентированных циклов (запрещена явная и неявная рекурсия в определениях).

В рассмотренной форме условие (3) принято в языке Клу. В языке Паскаль для определений рекурсивных типов введено более слабое условие корректности: любой ориентированный цикл в графе $G(D)$ должен содержать вершину, помеченную нетерминалом ⟨тип-указатель⟩, т.е. любая явная или неявная рекурсия в определении типа должна быть выражена через тип-указатель. В языке Алгол-68 условие (3) еще более ослаблено: определение вида считается корректным, если любой цикл в графе зависимостей определений видов содержит вершину, помеченную нетерминалом из множества D_1 = {⟨*имя*⟩, ⟨*процедура-без-параметров*⟩}, и вершину, помеченную нетерминалом из множества D_2 = {⟨*структура*⟩, ⟨*процедура-с-параметрами*⟩}. Принадлежность множеству D_1 определяет конечность памяти для представления вида, множеству D_2 – неприводимости вида самого к себе [55]. На практике условие корректности (3) встречается чаще в той же форме, что и в определении. Модификация алгоритма для языка Паскаль описана в работе [66].

Алгоритм семантического анализа системы определений. Задача алгоритма формулируется как вычисление всех семантических атрибутов, проверка корректности системы и выдача сообщений обо всех некорректных определениях. Идея алгоритма заключается в том, чтобы при построении графа зависимостей запоминать в нем план вычисле-

ния тех атрибутов выражений t_k, которые из-за неопределенности некоторых идентификаторов еще не могут быть вычислены. Алгоритм использует следующие структуры данных:

– таблицу определенных идентификаторов $D = \{(i_k, T_k)\}_{k=1,\ldots,n}$;

– таблицу использованных, но не определенных идентификаторов $N = \{(i_k, F_k)\}$;

– деревья вывода T_k;

– фиктивные деревья вывода F_k (из одного корня) для идентификаторов i_k, принадлежащих N (т.е. еще не определенных, но уже использованных).

Для выполнения алгоритма вводятся два дополнительных атрибута выражения T, где T совпадает с t_k или является его компонентой:

1) $a(T)$ – список всех использующих вхождений выражения T. Если T совпадает с t_k, то $a(X)$ есть список всех вершин, соответствующих использованиям идентификатора i_k (т.е. список всех использующих вхождений идентификатора i_k). Если же T является компонентой выражения t_k, то $a(T)$ – ссылка на непосредственного предка T;

2) $n(T)$ – число непосредственных компонент выражения T с еще не определенными атрибутами.

Алгоритм анализа работает следующим образом. Система определений анализируется слева направо. При этом строится граф зависимостей $G(D)$ и выполняются следующие действия, связанные с неопределенными идентификаторами:

1. При обработке первого использующего вхождения неопределенного идентификатора i_k для него строится фиктивное дерево вывода F_k; пара (i_k, F_k) добавляется в таблицу N, а первое вхождение i_k включается в список $a(F_k)$ его использующих вхождений.

2. При обработке использующего вхождение еще не определенного идентификатора i_k, уже включенного в N, либо вхождения недоопределенного идентификатора i_k (такого, что $n(T_k) > 0$), его вхождение включается в список использующих вхождений $a(T_k)$.

3. При обработке выражения T с недоопределенной компонентой S ($n(S) > 0$) устанавливается ссылка $a(S)$ из компоненты S на выражение T, а $n(T)$ увеличивается на единицу.

4. При обработке определения $i_k = T_k$ проверяется, не входит ли идентификатор i_k в таблицу N (если входит, то он исключается из нее). Пара (i_k, T_k) заносится в таблицу D (при этом контролируется неповторяемость определения i_k). Кроме того, если атрибуты T_k полностью вычислены ($n(T_k) = 0$), а список использующих вхождений $a(F_k)$ идентификатора i_k непуст, то запускается рекурсивная процедура просмотра списка использующих вхождений, которая завершает вычисление атрибутов для каждого элемента T списка $a(F_k)$ и запускает ту же процедуру просмотра списка использующих вхождений для $a(T)$. При вычислении атрибутов подвыражения T значение атрибута $n(T)$ обнуляется, а значение атрибута $n(a(T))$ уменьшается на единицу. Если при этом значение атрибута $n(T_k)$, где T_k – правая часть определения $i_k = T_k$, становится равным нулю, то запускается процедура просмотра списка $a(T_k)$, и т.п. Тем самым, процесс завершения вычисления атрибутов распространяется в направлениях, указываемых атрибутом a.

5. После завершения анализа системы определений D просматривается таблица N и для каждого ее элемента (i_k, F_k) выдается диагностика: "Идентификатор i_k не определен". Затем просматривается таблица $D = \{(i_k, T_k)\}$, и для идентификаторов i_k таких, что $n(T_k) > 0$, выдается диагностика: "Определение идентификатора i_k некорректно".

Доказательство корректности алгоритма и анализ его эффективности даны в работе [63]. Алгоритм анализирует систему определений D за один ее полный просмотр и один просмотр таблицы определений D (т.е. всех корней деревьев T_k). Атрибуты a и n обеспечивают отсутствие лишних визитов в вершины деревьев вывода, атрибуты которых еще не вычислены. Единственным серьезным допущением, неявно используемым в алгоритме, является возможность доступа в любой момент к произвольной компоненте любого дерева вывода T_k, т.е. необходимость хранения всех деревьев T_k, таблиц D и N одновременно в оперативной или виртуальной памяти. Иными словами, алгоритм рассчитан на ЭВМ с большим объемом памяти (например, на МВК "Эльбрус"). В остальном он не зависит от целевой ЭВМ.

Описанный метод анализа применен в первой версии системы Паскаль-Эльбрус, в системах Клу-Эльбрус и АБВ-Эльбрус.

П р и м е р. Рассмотрим работу алгоритма при анализе системы определений констант:

$a = b + c - d;$
$b = d * 2;$
$c = a * 5 * d;$
$d = 3;$
$e = f + 1.$

Семантическим атрибутом, подлежащим вычислению, является значение константы. На рис. 10 показаны упрощенные деревья вывода для правых частей определений идентификаторов: T_a, T_b, T_c, T_d, T_e. Стрелками

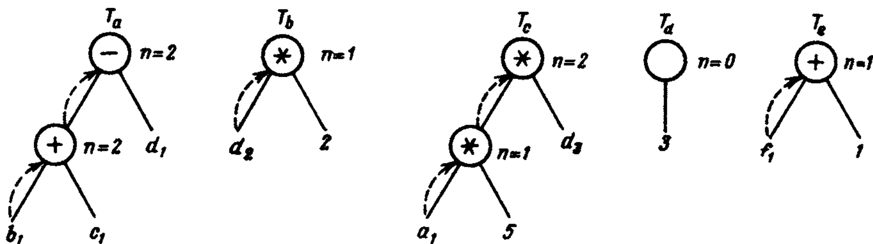


Рис. 10

указаны значения атрибута a для поддеревьев, индексами помечены номера использующих вхождений идентификаторов. В нетерминальных вершинах указаны значения атрибута n. Последовательность этапов работы алгоритма приведена в табл. 3. Показано состояние таблиц D и N после анализа соответствующего определения. В угловых скобках приведены списки использующих вхождений. Звездочками помечены деревья вывода с полностью вычисленными атрибутами.

Таблица 3. **Пример работы алгоритма семантического анализа системы определений**

Определение	D	N	Примечание
a	(a, T_a)	$(b, \langle b_1 \rangle)$ $(c, \langle c_1 \rangle)$ $(d, \langle d_1 \rangle)$	Создаются фиктивные деревья вывода для b, c и d
b	(a, T_a) $(b, T_b \langle b_1 \rangle)$	$(c, \langle c_1 \rangle)$ $(d, \langle d_1, d_2 \rangle)$	b переводится из N в D, но не может быть вычислен, так как $n(T_b) = 1$
c	$(a, T_a \langle a_1 \rangle)$ $(b, T_b \langle b_1 \rangle)$ $(c, T_c \langle c_1 \rangle)$	$(d, \langle d_1, d_2, d_3 \rangle)$	$n(T_c) = 2$
d	$(a, T_a \langle a_1 \rangle)$ (b, T_b^*) $(c, T_c \langle c_1 \rangle)$ (d, T_d^*)	*пусто*	Значение d вычисляется непосредственно, b – через список использующих вхождений
e	$(a, T_a \langle a_1 \rangle)$ (b, T_b^*) $(c, T_c \langle c_1 \rangle)$ (d, T_d^*) (e, T_e)	$(f, \langle f_1 \rangle)$	Диагностика: f не определен a определен некорректно c определен некорректно e определен некорректно

6.2.2. Универсальное представление строк. Для реализации языков символьной обработки одним из наиболее важных вопросов является выбор представления строк, которые в этих языках близки лисповским спискам. При представлении строк необходимо учесть специфику операций над ними в языках символьной обработки и обеспечить эффективность их реализации:

– как правило, обработка строк выполняется путем однократного или многократного просмотра строки; при анализе строка может разбиваться на части;

– наиболее часто используемые операции над строками – сцепление, вычисление длины строки и выделение подстрок (при рекурсивной структуре строки).

В качестве примера такого подхода можно рассматривать анализатор языка АБВ (§ 4.1).

Ввиду этих особенностей обработки строк для их наиболее рационального представления в языках символьной обработки не подходит не один из известных способов: связанное представление строк, при котором все символы строки располагаются подряд в одной области памяти (именно такое представление принято в зарубежных реализациях Снобола-4), не-

приемлемо из-за неэффективности выполнения основной операции сцепления (строки-аргументы копируются в новую область памяти), которая при большой длине строк (например, если в виде строк хранятся тексты программ или выражения для аналитических выкладок) приводит к быстрому исчерпанию памяти. С другой строны, чисто ссылочное представление (строка – список отдельных символов) для ЭВМ с большой разрядностью также приводит к неоправданным расходам памяти: например, на МВК "Эльбрус" при ссылочном представлении символ занимает 8 битов слова, ссылка – 20 битов, а 36 битов каждого слова теряются. Весьма искусственным представляется и дробление строк на фрагменты постоянной длины (например, по 8 символов), так как граница разбиения строки при анализе может не совпасть с границей фрагмента.

При создании первой версии системы АБВ-Эльбрус [41] было разработано и использовано универсальное представление строк для языков символьной обработки, обеспечивающее эффективность операций сцепления и анализа строк. Модификация этого представления использована в системе Снобол-Эльбрус [12].

Универсальное представление строк может быть применено в любом языке символьной обработки. Оно является компромиссом между противоречивыми требованиями удобства разбиения строки и экономии памяти для хранения ее символов. Рассмотрим две модификации представления строк, соответствующие двум вариантам семантики строк в языках символьной обработки:

– хранение строки в единственном экземпляре не требуется; строки могут дублироваться (АБВ);

– каждая строка хранится в единственном экземпляре, так как некоторые из них могут рассматриваться как уникальные имена атомов или переменных (Лисп, Снобол-4).

П р е д с т а в л е н и е с т р о к в с и с т е м е А Б В-Э л ь б р у с. Строка в языке АБВ может иметь древовидную структуру (п. 4.1.1). Основные характеристики строки – число элементов (ЧЭЛ) и число символов (ЧСИМ); элемент – это символ или подстрока. Область памяти для представления строк задается в виде двух векторов переменной длины: СТРОКИ (содержит списки компонент строк) и СИМВОЛЫ (содержат символы всех строк). Строка представляется паспортом строки – словом упакованной структуры с дополнительным тегом. Паспорт содержит поля СЧЭЛП – число элементов, СЧСИМП – число символов, СГОЛОВАП, СХВОСТП – ссылки на начало и конец списка компонент. Компонента строки – ссылка либо на группу смежных символов (поле СНАЧП – ссылка на начало группы в массиве СИМВОЛЫ, СДЛИНАП – длина группы), либо на подстроку (после СПОДСТРОКАП – ссылка на паспорт подстроки). Поле СЛЕДП – ссылка на следующий элемент. Поле СИМВП компоненты играет роль поля признака (0 – подстрока, 1 – группа символов). Представление строки

'ПРИМЕР'СЛОЖНОЙ'ДРЕВОВИДНОЙ'' СТРОКИ'

изображено на рис. 11.

При данном представлении операция сцепления строк реализуется установкой ссылки из конечной компоненты первой строки на начальную

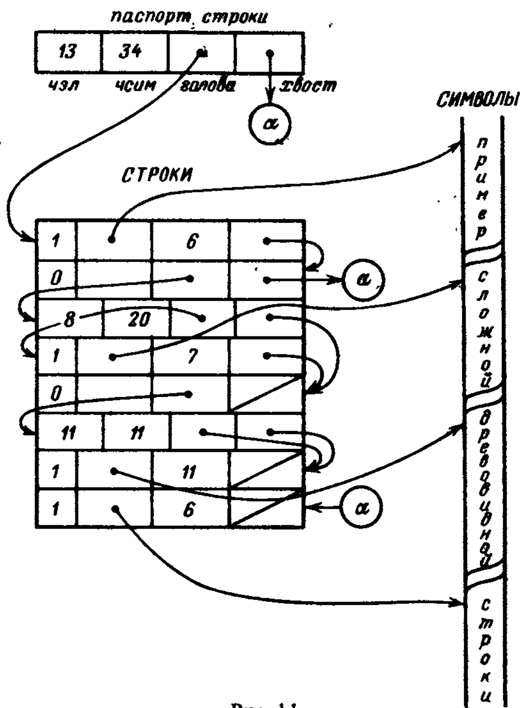


Рис. 11

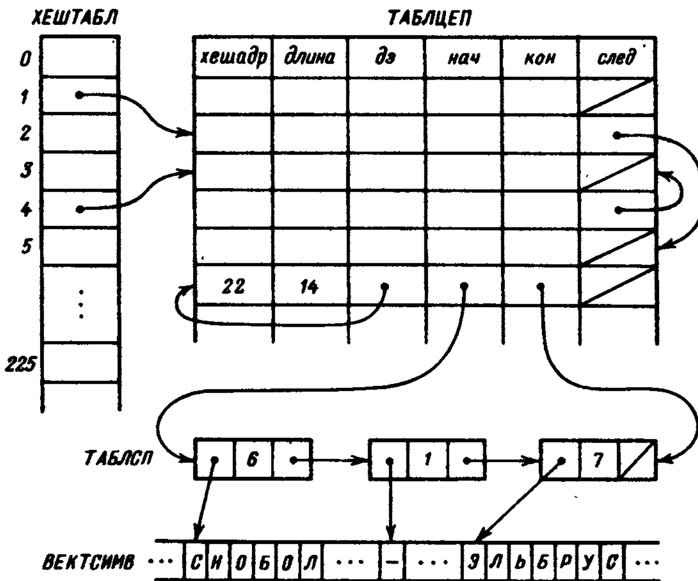


Рис. 12

компоненту второй строки и формированием паспорта результата по паспортам операндов. Выполнение операции анализа может привести к разбиению одной компоненты на две (при этом выполняется соответствующая реорганизация списка, не нарушающая корректности остальных ссылок). Тем самым, при однократном просмотре строки ни одна из операций не "портит" операндов. В случае необходимости сцепления компоненты строки с другой строкой (такие действия нарушают ссылочную структуру), эта компонента может быть скопирована операцией КОПИЯ. Забота о копировании возлагается на программиста.

Представление строк в системе Снобол-Эльбрус [12]. В отличие от языка АБВ семантика языка Снобол-4 требует хранения каждой строки в единственном экземпляре. Поэтому после формирования или ввода новой строки должен выполняться ее поиск в глобальной таблице строк, и лишь в случае его неудачи строка должна быть записана в таблицу. Для ускорения поиска наиболее удобно использовать функцию расстановки, поэтому в представлении строки должны храниться значения ее функции расстановки и связь в расстановочном поле. Кроме того, каждая строка может быть использована как идентификатор переменной, поэтому в таблице строк должно быть предусмотрено поле для значения этой переменной. Эти соображения положены в основу представления строк, разработанного Н.Н. Болдиновой для системы Снобол-Эльбрус. Представление изображено на рис. 12 (в качестве примера взята строка 'снобол–эльбрус'). Пример принадлежит Н.Н. Болдиновой. Смысл обозначений: ХЕШТАБЛ – оглавление расстановочного поля; ХЕШАДР – значение функции расстановки; ДЛИНА – длина строки; ДЗ – ссылка на значение строки как переменной; НАЧ – ссылка на начало списка; КОН – ссылка на конец списка; СЛЕД – связь в списке по функции расстановки; ТАБЛСП – вектор списков компонент; ВЕКТСИМВ – строка, содержащая символы всех строк.

Рассмотренное представление строк наиболее удобно для реализации операции сцепления, в которой при необходимости выполняется копирование первого операнда (если поле его конечной ссылки непусто). Таким образом, здесь забота о копировании с программиста снята. Перед выполнением операции сравнения с образцом строка преобразуется к линейному связному представлению, что несущественно увеличивает выполнение этой наиболее трудоемкой операции. Описанная реализация строк не зависит от целевой ЭВМ.

6.2.3. Реализация квазилокального контекста. Для реализации сложных динамических механизмов именования в интерпретационных языках на ЭВМ традиционной архитектуры используются, как правило, либо сложная техника ассоциативных списков, требующая поиска при интерпретации каждого использующего вхождения идентификатора, либо глобальная таблица текущих значений, для которой весьма трудоемким является процесс коррекции при вызовах подпрограмм. При реализации динамических языков в системе "Эльбрус", имеющей аппаратную поддержку алголоподобного процедурного контекста идентификаторов, возникает естественная идея попытаться свести к этим аппаратным средствам языковые механизмы идентификации и смены контекста, а включение необходимых интерпретационных вставок (например, для интерпретации пре-

рывания по неопределенному идентификатору) возложить на аппаратуру. Достаточным условием для возможности такой реализации аппарата имен является следующее свойство языка: для каждого идентификатора *а* должна быть статически определима цепочки вложенных блоков, в которых он потенциально может быть описан (хотя фактически некоторые из этих описаний, являющихся исполняемыми операторами, могут и не выполняться). Такое свойство идентификатора и самого аппарата имен в динамическом языке будем называть *квазилокальностью*. Свойством квазилокальности обладает аппарат имен языка АБВ (п. 4.1). Средства именования в языках Лисп и Снобол-4 не обладают этим свойством, так как в этих языках принята динамическая идентификация параметров и локальных переменных в подпрограммах и, кроме того, имеется возможность динамического создания нового идентификатора переменной.

Описывается метод эффективной реализации квазилокального контекста на МВК "Эльбрус", рассчитанный на использование в компиляторах [62]. Основные свойства предлагаемого метода:

– идентификация (поиск именуемого объекта по его квазилокальному идентификатору) реализуется командой *вел* (*лвел*), как и обращение к значению переменной в процедурных языках (п. 6.1.1); проверка необходимости поиска при идентификации и сам этот поиск выполняются аппаратно при исполнении той же команды *вел*;

– смена контекста при запуске, завершении, прерывании и прекращении действий (включая запуск процедурного атрибута составного объекта) выполняется обычными командами входа в процедуру, выхода из процедуры и перехода по метке.

Для пояснения метода рассмотрим пример программы на АБВ:

```
(ПОСЛ  (ВСИН Т1 и А)
       (ПОСЛ  (ВСИН Т2 и А)
              (ПОСЛ  (ВСИН Т3 и А)
                     (ВСИН Т4 и А)
                     (ВСИН Т5 и А) ... А % 1
                     (ЛИКВ и А) ... А
)))
```

В этой программе три вложенных последовательных действия, в каждом из которых могут выполняться операторы ВСИН со вторым операндом А. Будем обозначать эти операторы используемыми в них идентификаторами видов Т1 – Т5. Состояние стека, памяти и базовых регистров после выполнения оператора Т5 изображено на рис. 13.

При запуске последовательного действия уровня *ур* в памяти создается секция собственного контекста (п. 6.1.1), в которой квазилокальному идентификатору А соответствуют два слова:

об (*ур*, А) – информация о квазилокально именуемых объектах;
конт (*ур*, А) – информация о квазилокальном контексте А.

Слово *об* содержит: глобальный индекс (*инд*) идентификатора А, определяемый статически и используемый в качестве ключа при поиске атрибута; ссылку на активно именуемый объект (*акт*); ссылку в специальный пул *пас* на список пассивно именуемых объектов. Пул *пас* играет

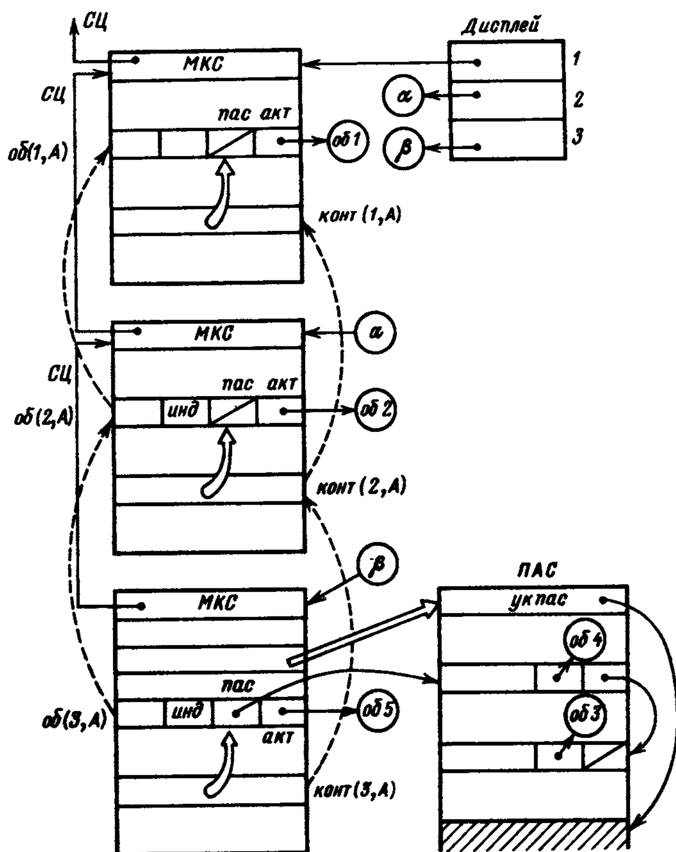


Рис. 13

роль динамической части секции контекста и создается по необходимости (в секции хранится заявка на память для пула).

Слово *конт* содержит дескриптор на часть секции контекста от дескриптора пула *пас* до слова *об*. Эта информация используется оператором ЛИКВ (п. 4.1.1).

Если в текущем последовательном действии идентификатор А еще не описан (оператор ВСИН не выполнен), то в слове *об* (*ур*, А) хранится адрес слова *об* (*ур*1, А), где *ур*1< *ур* – предыдущий уровень квазилокальности идентификатора А (на рис. 13 – пунктирные стрелки слева). Аналогично организован список слов *конт* (пунктирные стрелки справа). Если уровень квазилокальности А самый внешний, то в слове *об* и в слове *конт* первоначально хранятся технические метки процедур прерывания "неопределенный объект" и "неопределенный контекст".

Идентификация объекта реализована командами:

вел (*ур*, *смещ*$_A$); *впнн* (19, 20) ,

где команда *вел* считывает слово *об* (*ур*, А),*впнн* – выделяет поле *акт*. Если в программе были выполнены все операторы T1–T5, то в момент 1 (см. рис. 13) будет считано слово *об* (3, А), т.е. будет получен объект *об5*. Если же выполнен только оператор T1 или T2, то (после автоматического просмотра косвенной цепочки) будет считано слово *об* (1, А) или соответственно *об* (2, А) (найден объект *об1* или *об2*). Если же ни один из операторов T1–T5 не был выполнен, то после просмотра косвенной цепочки той же командой *вел* автоматически запускается техническая процедура прерывания "неопределенный объект". Аналогично реализован оператор ЛИКВ, который считывает слово *конт* (*ур*, А) командой *вел*, что приводит либо к загрузке в стек дескриптора на фактическую область контекста идентификатора А, либо к прерыванию. Смена квазилокального контекста при запуске последовательного действия и его завершения осуществляется аппаратно при выполнении команд входа и выхода. Прямой аппаратной поддержки не имеют только операторы ВСИН и ЛИКВ, которые реализуются несложной перестановкой ссылок. Выборка атрибута (составное обозначение) реализована аппаратной операцией линейного поиска по массиву.

В рассмотренном методе, по-видимому, впервые в практике разработки трансляторов для МВК "Эльбрус" удалось естественным образом соединить три аппаратных механизма различного назначения – процедурный контекст, косвенную адресацию и технические процедуры – для эффективной компиляционной реализации сложного динамического аппарата имен. Отметим также, что в данной реализации, в сравнении с реализацией [41], удалось последовательно провести принцип локализации информации о контексте, как это сделано в системе "Эльбрус" при аппаратной реализации процедур. Это и обеспечило эффективность реализации.

Для сравнения, в авторской реализации АБВ (системе АБВ-БЭСМ6 [70]) с каждым идентификатором связывается магазин именуемых объектов, который в процессе своего функционирования разбивается на части. Все операции над ним реализуются интерпретационными вставками. В описанной реализации аппарата имен АБВ для МВК "Эльбрус" списки слов *об* и *конт* могут рассматриваться также как одна из форм магазина имен, но операции над ним (в том числе и идентификация), благодаря возможностям МВК "Эльбрус", реализованы почти полностью аппаратно (интерпретационные действия при идентификации выполняются в процессе исполнения машинной команды *вел*).

6.2.4. Реализация абстрактных типов данных. Элементы современной концепции АТД, требования к базовому языку АТД, конструкции языка Клу и методы программирования на этом языке описаны в гл. 3. В заключение данной работы рассмотрим проблемы реализации языков с АТД и их решение в системе Клу-Эльбрус.

Проблемы реализации АТД. Реализация языковых механизмов АТД ставит перед разработчиками следующие проблемы.

1. Реализация вызова абстрактных операций. В описании реализации АТД могут присутствовать переменные и другие объекты периода исполнения. При вызове абстрактных операций должен быть установлен контекст реализации АТД, что сложно реализовать, если язык имеет динамические генераторы объектов (Симула-67, АБВ, Клу, Альфард) и, следова-

тельно, возможно создание произвольного числа экземпляров реализации. В этом случае для реализации АТД недостаточно чисто стекового механизма.

2. Реализация абстрактных исключительных ситуаций (Клу, Ада). Ситуации могут быть как программируемыми, так и стандартными. В трансляторе необходима реализация сложного алгоритма статического контроля распространения ситуаций [16], а в системе динамической поддержки должны быть средства поддержки динамических ситуаций.

3. Реализация итераторов (Клу, Альфард). При каждом запуске цикла, управляемого итератором, необходимо создавать новый экземпляр его собственных данных, включая неявную переменную, в которой хранится текущая точка возобновления итератора. Кроме того, необходимо обеспечить сопрограммную связь итератора с телом цикла. Метод реализации итераторов в системе Клу-Эльбрус описан в работе [40].

4. Реализация параметризованных модулей (Клу, Альфард, Ада). Наиболее распространенный способ реализации – макроподстановка: для каждой параметризации АТД с помощью смешанных вычислений генерируется новый экземпляр остаточной программы. Этот способ приводит к размножению копий близких программных модулей, что может вызвать резкое увеличение размера объектного кода. Для языка Клу применим другой способ: параметры рассматриваются как аргументы и передаются динамически. Такой способ реализации более прост, но несколько ухудшает эффективность программы.

5. Организация библиотеки модулей. Эта проблема характерна для всех языков модульного программирования. Библиотека должна обеспечить контроль межмодульных связей и пошаговую разработку программ из модулей. Для этого она должна содержать для каждого модуля, кроме его объектного кода, информацию периода компиляции о его интерфейсе, а для параметризованного модуля – заготовку его объектного кода со статической информацией о параметрах для контроля типов.

6. Реализация сборки программы для исполнения. В языках модульного программирования программа формируется из головного модуля и других модулей, явно или неявно связанных с головным отношением вызова. Традиционный метод сборки программы – статическая сборка, применяемая, например, в авторской реализации языка Модула-2 на *PDP*–11 [15]. Стандартные редакторы связей для статической сборки не подходят, так как при статической сборке требуется большой объем дополнительных действий по контролю типов, смешанным вычислением и генерации кода.

7. Использование фрагментов на других языках. Эта проблема – общая для всех языков программирования, так как в больших программах часто используются несколько языков (например, ассемблерные вставки в программе на языке высокого уровня). Для языков с АТД эта проблема формулируется в более систематической постановке: обеспечение возможности реализации АТД на другом языке и стыковки ее с интерфейсом, записанным на базовом языке АТД.

8. Усложнение семантического контроля. Языки с АТД ставят проблему реализации значительно более сложных алгоритмов статического контро-

ля, чем в традиционных языках:

– трансляция полиморфных операций (например, трактовка инфиксной формы $a + b$ как $t \$ add(a, b)$ в языке Клу);

– проверка корректности параметризации АТД (например, проверка выполнения ограничений на параметр-тип в языке Клу);

– статический контроль распространения ситуаций [16];

– конкретизация типа при контроле типов операций параметризованного модуля.

Рассмотренные сложные проблемы 1)–8) реализации АТД большинство разработчиков трансляторов решает на ЭВМ традиционной архитектуры, на которых эффективная реализация рассмотренных элементов АТД в полном объеме вызывает большие трудности. Например, такие элементы, как вызов абстрактных операций и распространение динамических ситуаций, на традиционных ЭВМ интерпретируются; для решения проблем сборки, организации библиотеки модулей и программирования реализаций АТД на других языках отсутствует единая архитектурная основа.

Архитектурная поддержка АТД в системе "Эльбрус". Для унифицированного подхода к реализации АТД в системе "Эльбрус" используются следующие элементы архитектуры МВК "Эльбрус":

– аппаратная поддержка процедур с собственным контекстом (п. 6.1.1) и реализованные на ее основе модульные объекты (п. 1.31);

– аппаратная поддержка исполнения программы, состоящей из нескольких файлов объектного кода (или связи модулей);

– системная поддержка динамических ситуаций (п. 6.1.1);

– аппаратная реализация технических процедур (используется для задержанной генерации модульных объектов);

– средства работы с архивом в системе файлов (используются для представления информации в библиотеке модулей);

– унифицированная структура РФОК – кодового и символьно-табличного представления программы (используется для представления информации о модулях, их объектном коде и интерфейсе).

Реализация языка Клу для МВК "Эльбрус". Рассмотренные проблемы реализации АТД решены в системе Клу-Эльбрус (см. гл. 3) с применением перечисленных элементов архитектуры МВК "Эльбрус".

Транслятор в системе Клу-Эльбрус работает по двупросмотровой схеме. На первом просмотре осуществляется синтаксический анализ, полный семантический контроль; программа переводится в промежуточную форму (дерево).

Для трансляции выражений применяется новый метод смешанного анализа [17], основанный на использовании аппаратного стека выражений в качестве магазина МП-автомата. Для трансляции описаний-тождеств применяется модификация алгоритма анализа системы определений (п. 6.2.1).

На втором просмотре генерируется объектный код. Каждый модуль (процедура, кластер, итератор) отображается в стандартный расширенный файл объектного кода модульного объекта. Полученный файл записывается в библиотеку модулей.

Реализация модулей Клу. Все элементы кластера, несмотря на некоторое различие подходов к АТД в языках Клу и Эль-76 (см. гл. 3), адекватно отображаются в структуру модульного объекта МВК "Эльбрус". В интерфейс кластера (как модульного объекта) входят метки операций; для параметризованных кластеров – также значения параметров. Реализацию модульного объекта, соответствующего кластеру, составляют метки скрытых подпрограмм и адреса собственных переменных в глобальной области памяти. Взаимодействие модулей Клу (например, при вызове операции кластера из другого модуля) реализуется стандартными средствами работы с модульными объектами: подключение кластера – вызовом системной процедуры *ген*, создающей экземпляр модульного объекта; вызов операции – индексацией области его интерфейса и входом в процедуру по полученной метке операции интерфейса.

Несколько более сложна реализация собственных переменных. В соответствии с семантикой языка Клу они размещаются в глобальной области памяти и создаются при первом вызове содержащего их модуля. При подключении модуля процедурой *ген*, при инициализации области памяти реализации выполняется ассоциативный поиск области его собственных переменных в глобальном контексте, в котором исполняются все модули клу-программы. Адреса собственных переменных записываются в область реализации модуля. Такой способ реализации уникальной статической области памяти является вынужденной мерой, так как система "Эльбрус" полностью ориентирована на динамическое распределение памяти, и сформировать адреса собственных переменных во время трансляции невозможно.

Все модули клу-программы исполняются в контексте *ядра* клу-системы, которое также реализовано в виде модульного объекта. Экземпляр ядра создается при запуске клу-программы. В терминах модульных объектов (п. 1.31), модуль ядра является *формальным контекстом* всех модулей, полученных в результате трансляции с языка Клу. Ядро содержит метки стандартных процедур, ссылки на файлы ввода-вывода и дескриптор глобальной области памяти, в которой размещаются собственные переменные всех модулей, составляющих клу-программу.

Динамическая и статическая сборка. Описанным способом реализуется режим динамической сборки программы в системе Клу-Эльбрус. Как отмечается в п. 3.11, в целях повышения эффективности исполнения больших программ на языке Клу введен также режим статической сборки. Модуль, компилируемый в режиме ¤ *уст стат* для последующей статической сборки (п. 3.11), транслируется не в объектный код, а в дерево программы, которое записывается в библиотеку вместо файла объектного кода. Статическая сборка включает следующие действия:

– определения всех конкретных модулей и их параметризаций, участвующих в выполнении программы;

– генерация кода программных сегментов для этих модулей и их объединение в один файл объектного кода;

– формирование адекватной структуры ДФОК, обеспечивающей стыковку с ОСПО (динамическую диагностику в терминах текста и комплексацию);

– формирование прообраза "статической" области памяти для собственных переменных всех конкретных модулей программы (обращение к ним

во время исполнения в этом режиме реализуется обычными командами, по адресной паре);

– нумерация обозначений типов, используемых в данной программе при присваивании переменной типа any (в объектном коде эти типы будут представлены целочисленными номерами).

Реализация библиотеки модулей. Библиотека модулей системы Клу-Эльбрус (см. п. 3.11) не являетя какой-либо специализированной структурой во внешней памяти, используемой и "понятной" лишь одной клу-системе. Все компоненты библиотеки (информация о модулях и их контекстные связи) реализованы в виде элементов общего архива системы "Эльбрус". Вся информация о модуле представлена в стандартном виде в его расширенном файле объектного кода. Каждому модулю соответствует РФОК интерфейса и один или несколько РФОК реализации. Ссылка на контекст компиляции модуля реализована через атрибут *внешконт* (внешний контекст) файла объектного кода. Контексту компиляции также соответствует РФОК, в локальном словаре которого хранится информация об образующих данный контекст описаниях-тождествах и внешних именах модулей. Таким образом, все компоненты библиотеки модулей Клу "понятны" многоязыковым компонентам ОСПО, что обеспечивает адекватную динамическую диагностику, комплексацию и другие услуги, предоставляемые общими средствами языковой поддержки ОСПО.

Использование реализаций на других языках. Реализация любого модуля Клу в системе Клу-Эльбрус может быть написана на другом языке (например, на языке Эль-76) и оттранслирована соответствующим компилятором в РФОК реализации. После этого должна быть вызвана клу-система в специальном режиме, которая по информации в ДФОК интерфейса и реализации определяет их совместимость по данным. Основой унификации представления данных является аппаратное представление векторов и простых объектов. При стыковке интерфейса на языке Клу и реализации на языке Эль-76 они считаются совместимыми по данным, если соответствующие элементы интерфейса и реализации (аргументы и результаты процедур) имеют одинаковый формат. Проблем стыковки по управлению не возникает благодаря аппаратной реализации процедур в системе "Эльбрус".

Итак, в данной главе рассмотрены внутренние аспекты архитектуры системы "Эльбрус", доступные лишь системному программисту – разработчику трансляторов. Показано, что аппаратура и ОС МВК "Эльбрус" обеспечивает поддержку реализации всех основных элементов современных языков программирования, включая как более традиционные процедурные языки, так и динамические (интерпретационные), языки модульного программирования и языки с АТД. На основе возможностей архитектуры системы "Эльбрус" разработан ряд новых методов реализации и осуществлена разработка системы программирования Паскаль, Клу, АБВ, Снобол-4, Рефал, Форт, ДИАШАГ и Модула-2. Эти результаты являются подтверждением перспективности современного подхода к архитектуре ЭВМ – ее ориентации на языки и методы программирования, – практически воплощенного в системе "Эльбрус".

ПРИЛОЖЕНИЕ 1

СИНТАКСИС СТАНДАРТНОГО ЯЗЫКА ПАСКАЛЬ И ЯЗЫКА ПАСКАЛЬ-ЭЛЬБРУС

Обозначения:

Метасимвол	Смысл
=	есть по определению
|	либо
[*x*]	0 или 1 вхождение *x*
{*x*}	0 или более вхождений *x*
(*x* | *y*)	обязательное вхождение *x* или *y*
«*xyz*» или "*xyz*"	терминальный символ *xyz*
метаслово	нетерминальный символ

Метаслово – последовательность букв и дефисов, начинающаяся с буквы.

Синтаксис:

базовый-тип = порядковый тип
блок = раздел-описаний-меток
раздел-определений-констант
раздел-определений-типов
раздел-описаний-переменных
раздел-описаний-процедур-и-функций
раздел-операторов
блок-программы = блок
блок-процедуры = блок
блок-функции = блок
буква = «*a*» | «*b*» | «*c*» | «*d*» | «*e*» | «*f*» | «*g*» |
«*h*» | «*i*» | «*j*» | «*k*» | «*l*» | «*m*» | «*n*» |
«*o*» | «*p*» | «*q*» | «*r*» | «*s*» | «*t*» | «*u*» |
«*v*» | «*w*» | «*x*» | «*y*» | «*z*».
буферная-переменная = переменная-файл «↑»
вариант = список-констант-выбора «:» «(« список-полей »)»
вещественное-без-знака =
целое-без-знака «.» дробная-часть [«e» порядок] |
целое-без-знака «e» порядок

вещественное-со-знаком = [знак] вещественное-без-знака
выборка-поля = переменная-запись «.» обозначение-поля |
идентификатор-выборки-поля
вызов-процедуры = идентификатор-процедуры (
[список-фактических-параметров] | список-параметров-*read* |
[список-параметров-*readln*] | [список-параметров-*write*] |
[список-параметров-*writeln*])
вызов-функции = идентификатор-функции
[список-фактических-параметров]
выражение = простое-выражение [операция-отношения
простое-выражение]
директива = буква {(буква | цифра)}
дробная-часть = последовательность-цифр
заголовок-программы =
«*program*» идентификатор [«(» параметры-программы «)»]
заголовок-процедуры =
«*procedure*» идентификатор
[список-формальных-параметров]
заголовок-функции = «*function*» идентификатор
[список-формальных-параметров] «:» тип-результата
знак = «+» | «–»
идентификатор = буква {(буква | цифра)}
идентификатор-выборки-поля = идентификатор
идентификатор-константы = идентификатор
идентификатор-переменной = идентификатор
идентификатор-поля = идентификатор
идентификатор-порядкового-типа = идентификатор-типа
идентификатор-простого-типа = идентификатор-типа
идентификатор-процедуры = идентификатор
идентификатор-сложного-типа = идентификатор-типа
идентификатор-типа = идентификатор
идентификатор-типа-вещественное = идентификатор-типа
идентификатор-типа-указатель = идентификатор-типа
идентификатор-функции = идентификатор
изображение-порядкового-типа =
тип-перечисление | тип-диапазон
изображение-простого-типа =
изображение-порядкового-типа | изображение-вещественного-типа
изображение-сложного-типа =
«packed» неупакованный-сложный-тип
изображение-типа = изображение-порядкового-типа |
изображение-сложного-типа | изображение-типа-указатель
изображение-типа-указатель = «↑» тип-области
иначе = «else» оператор
индекс-выбора = выражение
индексированная-переменная =
переменная-массив «[» индексное выражение
{«,» индексное-выражение} «]»

индексное-выражение = выражение
константа = [знак] (число-без-знака | идентификатор-константы)
| строка-символов
константа-без-знака = число-без-знака | строка-символов |
идентификатор-константы | ⟨⟨ nil ⟩⟩
константа-выбора = константа
конструктор-множества =
⟨⟨ [⟩⟩ [обозначение-элемента {⟨⟨ , ⟩⟩
обозначение-элемента }] ⟨⟨] ⟩⟩
косвенная-переменная = переменная-указатель ⟨⟨ ↑ ⟩⟩
логическое-выражение = выражение
метка = последовательность-цифр
множитель =
обозначение-переменной | константа-без-знака |
идентификатор-границы | вызов-функции |
конструктор-множества | ⟨⟨ (⟩⟩выражение ⟨⟨) ⟩⟩ |
⟨⟨ not ⟩⟩ множитель
начальное-значение = выражение
неупакованная-схема-формальных-массивов =
⟨⟨ array ⟩⟩ ⟨⟨ [⟩⟩ спецификация-типа-индекса
{⟨⟨ , ⟩⟩ спецификация-типа-индекса } ⟨⟨] ⟩⟩ ⟨⟨ of ⟩⟩
(идентификатор-типа | схема-формальных-массивов)
неупакованный-сложный-тип =
тип-массив | тип-запись | тип-множество | тип-файл
обозначение-переменной =
полная-переменная | компонентная-переменная |
косвенная-переменная | буферная-переменная
обозначение-поля = идентификатор-поля
обозначение-процедуры = "*procedure*" идентификатор-процедуры
обозначение-типа = идентификатор-типа | изображение-типа
обозначение-функции = "*function*" идентификатор-функции
обозначение-элемента = выражение " . . " выражение
общая-часть = секция-записи { " ; " секция записи }
оператор = метка " : " (простой-оператор | сложный-оператор)
оператор-выбора = "*case*" индекс-выбора "*of*"
элемент-списка-выбора { " ; " элемент-списка-выбора } [" ; "]
"*end*"
оператор-для =
"*for*" управляющая-переменная ":=" начальное-значение
("*to*" | " *downto*") конечное-значение
"*do*" оператор
оператор-если =
"*if*" логическое-выражение "*then*" оператор [иначе]
оператор-над-записями =
"*with*" список-переменных-записей "*do*" оператор
оператор-перехода = "*goto*" метка
оператор-повтор = "*repeat*" последовательность-операторов
"*until*" логическое выражение

оператор-пока =

"*while*" логическое-выражение "*do*" оператор

оператор-присваивания =

(обозначение-переменной | идентификатор-функции)

":=" выражение

оператор-цикла =

оператор-повтор | оператор-пока | оператор-для

операции-отношения =

"=" | "< >" | "<" | ">" | "<=" | ">=" | "*in*"

операция-типа-сложения "+" | "–" | "*or*"

операция-типа-умножения = "*" | "/" | "*div*" | "*mod*" | "*and*"

описание-переменных =

список-идентификаторов ":" обозначение-типа

описание-процедуры =

заголовок-процедуры ";" директива |

обозначение-процедуры ";" блок-процедуры |

заголовок-процедуры ";" блок-процедуры

описание-функции =

заголовок-функции ";" директива |

обозначение-функции ";" блок-функции |

заголовок-функции ";" блок-функции

определение-константы = идентификатор "=" константа

определение-типа = идентификатор "=" обозначение-типа

параметры-программы = список-идентификаторов

параметр-*write* =

выражение [":" выражение [":" выражение]]

переменная-запись = обозначение-переменной

переменная-массив = обозначение-переменной

переменная-указатель = обозначение-переменной

переменная-файл = обозначение-переменной

поле-признака = идентификатор

полная-переменная = идентификатор-переменной

порядковый-тип =

изображение-порядкового-типа |

идентификатор-порядкового-типа

порядок = целое-со-знаком

последовательность-операторов = оператор { ";" оператор}

последовательность-цифр = цифра { цифра }

представление-апострофа = "''"

признак-варианта = [поле-признака ":"] тип-признака

программа = заголовок-программы ";" блок-программы

простое-выражение =

[знак] терм { операция-типа-сложения терм }

простой-оператор =

пустой-оператор | оператор-присваивания |

вызов-процедуры | оператор-перехода

простой-тип =

порядковый-тип | идентификатор-вещественного-типа

пустой-оператор =
раздел-операторов = составной-оператор
раздел-описаний-меток =
["*label*" метка { " , "метка } " ; "]
раздел-описаний-переменных =
["*var*" описание-переменных " ; " { описание-переменных
" ; " }]
раздел-описаний-процедур-и-функций =
{(описание-процедуры | описание-функции) " ; " }
раздел-определений-констант =
["*const*" определение-константы " ; "
{определение-константы " ; " }]
раздел-определений-типов =
["*type*" определение-типа " ; " { определение-типа " ; " }]
секция-записи = список-идентификаторов " : " обозначение-типа
секция-формальных-параметров =
спецификация-параметров-значений |
спецификация параметров-переменных |
спецификация-параметра-процедуры |
спецификация-параметра-функции |
спецификация-формальных-параметров-массивов
символ-строки =
один из-набора-символов-определяемых-реализаций
сложный-оператор =
составной-оператор | условный-оператор |
оператор-цикла | оператор-над-записями
сложный-тип =
изображение-сложного-типа | идентификатор-сложного-типа
служебное-слово =
"*and*" | "*array*" | "*begin*" | "*case*" | "*const*" |
"*div*" | "*do*" | "*downto*" | "*else*" | "*end*" | "*file*" |
"*for*" | "*function*" | "*goto*" | "*if*" | "*in*" | "*label*" |
"*mod*" | "*nil*" | "*not*" | "*of*" | "*or*" | "*packed*" |
"*procedure*" | "*program*" | "*record*" | "*repeat*" |
"*set*" | "*then*" | "*to*" | "*type*" | "*until*" | "*var*" |
"*while*" | "*with*"
сменная-часть =
"*case*" признак-варианта "*of*" вариант {" ; " вариант}
составной-оператор =
"*begin*" последовательность-операторов "*end*"
спецификация-параметра-процедуры = заголовок-процедуры
спецификация-параметра-функции = заголовок функции
спецификация-параметров-значений =
список-идентификаторов " : " идентификатор-типа
спецификация-параметров-переменных =
"*var*" список-идентификаторов " : " идентификатор-типа
спецификация-типа-индекса =
идентификатор " . . " идентификатор " : "
идентификатор-порядкового-типа

спецификация-формальных-массивов-значений =
список-идентификаторов ":" схема-формальных-массивов
спецификация-формальных-массивов переменных =
"*var*" список-идентификаторов ":"
схема-формальных-массивов
спецификация-формальных-параметров-массивов =
спецификация-формальных-массивов-значений |
спецификация-формальных-массивов-переменных
спецсимвол =
"+" | "–" | "*" | "/" | "=" | "<" | ">" | "[" |
"]" | "." | "," | ":" | ";" | "↑" | "(" | ")" |
"<>" | "<=" | ">=" | ":=" | ".." | служебное-слово
список-идентификаторов = идентификатор {"," идентификатор}
список-констант-выбора = константа-выбора {"," константа-выбора}
список-параметров-read =
"(" [переменная-файл ","] обозначение-переменной
{"," обозначение-переменной } ")"
список-параметров-*readln* =
["(" [переменная-файл ","] обозначение-переменной
{"," обозначение-переменной } ")"]
список-параметров-*write* =
"(" [переменная-файл ","] параметр-*write*
{"," параметр-*write* } ")"
список-параметров-*writeln* =
["(" [переменная-файл ","] параметр-*write*
"," параметр-*write* ")"]
список-переменных-записей =
переменная-запись {"," переменная-запись}
список-полей =
[(общая-часть [";" сменная-часть] |
сменная-часть) [";"]]
список-фактических-параметров =
"("фактический-параметр {"," фактический-параметр}
")"
список-формальных-параметров =
"(" секция-формальных-параметров
{"," секция-формальных-параметров } ")"
строка-символов = "'" элемент-строки {элемент-строки} "'"
схема-формальных-массивов =
упакованная-схема-формальных-массивов |
неупакованная-схема-формальных-массивов
терм = множитель {операция-типа-умножения множитель}
тип-диапазон = константа ".." константа
тип-запись = "*record*" список-полей "*end*"
тип-индекса = порядковый-тип
тип-компонент = обозначение-типа
тип-массив =
"*array*" "["тип-индекса {"," тип-индекса } "]"
"*of*" тип-компонент

тип-множество = "*set*" "*of*" базовый-тип
тип-перечисление = "(" список-идентификаторов ")"
тип-признака = идентификатор-порядкового-типа
тип-результата =
    идентификатор-простого-типа |
    идентификатор-типа-указатель
тип-указатель =
    изображение-типа-указатель |
    идентификатор-типа-указатель
тип-файл = "*file*" "*of*" тип-компонент
упакованная-схема-формальных-массивов =
    "*packed*" "*array*" "["спецификация-типа-индекса
    "]" "*of*" идентификатор-типа
управляющая-переменная = полная-переменная
условный-оператор = оператор-если | оператор-выбора
фактический-параметр =
    выражение | обозначение-переменной |
    идентификатор-процедуры | идентификатор-функции
целое-без-знака = последовательность-цифр
целое-со-знаком = [знак] целое-без-знака
цифра = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
число-без-знака =
    целое-без-знака | вещественное-без-знака
число-со-знаком =
    целое-со-знаком | вещественное-со-знаком
элемент-списка-выбора =
    список-констант-выбора ":" оператор
элемент-строки =
    представление-апострофа | символ-строки

Примечание. Лексика и синтаксис входного языка Паскаль-Эльбрус имеют следующие особенности:

1) буквы верхнего и нижнего регистра не различаются;

2) разрешены русские буквы в идентификаторах, строках символов и комментариях;

3) введена русская лексика;

4) вместо символа "↑" используется символ "@".

ПРИЛОЖЕНИЕ 2

РУССКАЯ ЛЕКСИКА ЯЗЫКА ПАСКАЛЬ-ЭЛЬБРУС

П.2.1. Служебные слова

Английская мнемоника	Русская мнемоника	Английская мнемоника	Русская мнемоника
and	*и*	*not*	*не*
array	*массив*	*of*	*из*
begin	*начало*	*or*	*или*
case	*выбор*	*packed*	*упак*
const	*конст*	*procedure*	*процедура*
div	*дел*	*program*	*программа*
do	*исп, цикл*	*record*	*запись*
downto	*внизд*о	*repeat*	*повтор*
else	*иначе*	*set*	*множ*
end	*конец*	*then*	*то*
file	*файл*	*to*	*по*
for	*для*	*type*	*тип*
function	*функция*	*until*	*до*
goto	*на*	*var*	*перем*
if	*если*	*while*	*пока*
in	*элем*	*with*	*над*
label	*метка*		
mod	*мод*		
nil	*нич*		

Примечание: служебное слово *do* используется в языке Паскаль в двух контекстах: в операторах цикла и в операторах над записями. В певом случае в русском варианте рекомендуется использовать служебное слово *цикл*, во втором – *исп*. Оба эти служебных слова равноправны.

П.2.2. Предопределенные идентификаторы

Английская мнемоника	Русская мнемоника	Английская мнемоника	Русская мнемоника
abs	*абс*	*eof*	*конф*
arctan	*арктанг*	*eoln*	*конс*
boolean	*лог*	*exp*	*эксп*
char	*лит*	*false*	*ложь*
chr	*лт*	*get*	*взять*
cos	*кос*	*input*	*ввод*
dispose	*осв*	*integer*	*цел*
ln	*лн*	*rewrite*	*стереть*
maxint	*максцел*	*round*	*окр*
new	*нов*	*sin*	*син*
odd	*нечет*	*sqr*	*квадрат*
ord	*номер*	*sqrt*	*корень*
output	*вывод*	*succ*	*след*
pack	*пак*	*text*	*текст*
page	*стран*	*true*	*истина*
pred	*пред*	*trunc*	*целч*
put	*выдать*	*unpack*	*распак*
read	*чит*	*write*	*печ*
readln	*читс*	*writeln*	*печс*
real	*вещ*	*forward*	*ниже*
reset	*считать*	*external*	*внеш*

П.2.3. Дополнительные предраспределенные идентификаторы входно языка Паскаль-Эльбрус

Английская мнемоника	Русская мнемоника	Английская мнемоника	Русская мнемоника
card	*мощн*	*date*	*дата*
maxel	*максэл*	*time*	*время*
clock	*часы*	*second*	*секунда*

СИНТАКСИС ЯЗЫКА КЛУ И ЯЗЫКА КЛУ-ЭЛЬБРУС

Обозначения. Синтаксис описывается с помощью расширенной формы Бэкуса–Наура. Правило в этой форме имеет следующий вид:

```
нетерминал :: =   альтернатива
                 | алььернатива . . .
                 | альтернатива
```

Используются следующие расширения БНФ:

1) *a*, . . . – список из одного или более вхождений нетерминала *a*, разделенных запятыми;

2) { *a* } – последовательность из одного или более вхождений нетерминала *a* либо пустая последовательность;

3) [**a**] – возможное (необязательное) вхождение нетерминала *a*: пусто либо *a*.

Символ "точка с запятой", используемый в описании синтаксиса, в языке Клу носит необязательный характер, но для простоты его вхождения обозначены "; ", а не [*; *]. Нетерминальные символы набраны обычным шрифтом, служебные слова – полужирным.

```
модуль :: =  { тождество } процедура
            | { тождество } итератор
            | { тождество } кластер
процедура ::=
        ид = proc [*параметры*] аргументы [*результаты*]
                  [*ситуации*] [*ограничения*];
        тело-подпрограммы
         end ид;
итератор ::=
     ид = iter [*параметры*] аргументы [*выдача*]
              [*ситуации*] [*ограничения*];
                тело-подпрограммы
       end ид;
```

```
кластер ::=
    ид = cluster [*параметры*] is ид, . . .
                [*ограничения*] ;
                  тело-кластера
    end ид;
параметры ::= [параметр, . . .]
параметр ::=  ид, . . . : type
             | ид, . . . : обозначение-типа
аргументы ::= ([*описание, . . .*])
описание ::=  ид, . . . : обозначение-типа
результаты ::= returns (обозначение-типа, . . .)
выдача ::= yields (обозначение-типа, . . .)
ситуации ::= signals (ситуация, . . . )
ситуация ::= имя [*(обозначение-типа, . . .)*]
ограничения ::= where ограничение, . . .
ограничение ::= ид has описание-операций
              | ид in множество-типов
множество-типов ::=
          {ид | ид has описание-операций, . . .; {тождество}}
          | ид
описание-операций ::=
        имя-операции, . . . : обозначение-типа
имя-операции ::= имя [*[константа, . . .]*]
константа ::= выражение | обозначение-типа
тело-подпрограммы ::=
          {тождество}
          {собственные-переменные}
          {оператор}
тело-кластера ::=
          {тождество}
             rep = обозначение-типа
          {тождество}
          {собственные-переменные}
           подпрограмма {подпрограмма}
подпрограмма ::=
           процедура | итератор
тождество ::=
           ид = константа;
          | ид = множество-типов;
собственные-переменные ::=
      own   описание;
      own   ид: обозначение-типа := выражение;
      own   описание, . . . := вызов;
обозначение-типа ::=
      null
```

```
      | bool
      | int
      | real
      | char
      | string
      | any
      | rep
      | cvt
      | array          [обозначение-типа]
      | sequence       [обозначение-типа]
      | record         [обозначение-поля, . . . ]
      | struct         [обозначение-поля, . . . ]
      | oneof          [обозначение-поля, . . . ]
      | variant        [обозначение-поля, . . . ]
      | proctype       ( [* обозначение-типа, . . . *] )
                       [* результаты *]  [* ситуации *]
      | itertype       ( [* обозначение-типа, . . . *] )
                       [* выдача *]  [* ситуации *]
      | ид             [константа, . . . ]
      | ид
обозначение-поля ::=
          имя, . . . : обозначение-типа
оператор ::=
          описание
        | ид: обозначение-типа := выражение;
        | описание, . . . := вызов;
        | ид, . . . := вызов;
        | ид, . . .  := выражение, . . . ;
        | первичное. имя := выражение;
        | первичное [выражение] := выражение;
        | вызов;
        | while выражение do тело end;
        | for  [* описание, . . . *]  in вызов do тело end;
        | for  [* ид, . . . *]  in вызов do тело end;
        | if выражение then тело
            { elseif выражение then тело }
          [* else тело *]
            end;
      | tagcase выражение
          { помеченная-альтернатива }
          { помеченная-альтернатива }
           [* others : тело *]
      end;
     | return [* (выражение, . . . ) *] ;
     | yield   [* (выражение, . . . ) *] ;
     | signal  имя [* (выражение, . . . ) *] ;
     | exit  имя [* (выражение, . . . ) *] ;
     | break;
     | continue;
```

| **begin** тело **end**;
| оператор **resignal** имя, . . .
| оператор **except** { реакция-при }
[* реакция-прочие *]
end
помеченная-альтернатива ::=
tag имя, . . . [*(ид: обозначение-типа)*] : тело
реакция-при ::=
when имя, . . . [*(описание, . . .) *] : тело
| **when** имя, . . . (*) : тело
реакция-прочие ::=
others [*(ид: обозначение-типа)*] : тело
тело ::=
{ тождество }
{ оператор }
выражение ::=
| первичное
| (выражение)
| ~ выражение % приоритет = 6
| – выражение % 6
| выражение ** выражение % 5
| выражение // выражение % 4
| выражение / выражение % 4
| выражение * выражение % 4
| выражение || выражение % 3
| выражение + выражение % 3
| выражение – выражение % 3
| выражение < выражение % 2
| выражение <= выражение % 2
| выражение = выражение % 2
| выражение >= выражение % 2
| выражение > выражение % 2
| выражение ~ < выражение % 2
| выражение ~ <= выражение % 2
| выражение ~ = выражение % 2
| выражение ~ >= выражение % 2
| выражение ~ > выражение % 2
| выражение & выражение % 1
| выражение **cand** выражение % 1
| выражение | выражение % 0
| выражение **cor** выражение % 0
первичное ::=
nil
| **true**
| **false**
| изображение-целого
| изображение-вещественного
| изображение-символа

| изображение-строки
| ид
| ид [константа, . . .]
| первичное. имя
| первичное [выражение]
| вызов
| обозначение-типа $ {поле, . . .}
| обозначение-типа $ [[* выражение: *] [* выражение, . . . *]]
| обозначение-типа $ имя [*[константа, . . .]*]
| **force** обозначение-типа
| **up** (выражение)
| **down** (выражение)
вызов ::= первичное ([* выражение, . . . *])
поле ::= имя, . . . : выражение

Служебное слово – любой из идентификаторов, выделенных полужирным шрифтом в описании синтаксиса. Буквы верхнего и нижнего регистра в служебных словах не различаются.

Имя, ид (*идентификатор*) – последовательность букв, цифр и символов "подчерк" (_), начинающаяся с буквы или подчеркивания и не являющаяся служебным словом. Буквы верхнего и нижнего регистра в именах и идентификаторах не различаются.

Изображение целого – последовательность из одной или более десятичных цифр.

Изображение вещественного – мантисса, из которой может следовать порядок. *Мантисса* – последовательность из одной или более десятичных цифр либо две последовательности цифр, соединенные точкой (одна из них может быть пустой). Мантисса должна содержать по крайней мере одну цифру. *Порядок* – символ "Е" или "е", за которым может следовать знак и далее последовательность из одной или более десятичных цифр. Если мантисса не содержит точку, порядок обязателен.

Изображение символа – либо заключенный в одиночные кавычки печатный символ ASCII (с восьмеричным кодом от 40 до 176), отличный от одиночной кавычки и обратной косой черты, либо одна из следующих заменяющих последовательностей, заключенная в одиночные кавычки:

заменяющая последовательность	символ
\'	' (одиночная кавычка)
\"	" (двойная кавычка)
\\	\ (обратная косая черта)
n	перевод строки
t	горизонтальная табуляция
p	переход на новую страницу
b	возврат на один символ
r	возврат каретки
v	вертикальная табуляция
***	символ, заданный восьмеричным кодом из трех цифр

Заменяющие последовательности могут быть записаны с использованием букв верхнего регистра.

Изображение строки – последовательность представлений символов (возможно, пустая), заключенная в двойные кавычки. *Представление символа* – либо печатный символ ASCII, отличный от двойной кавычки или обратной косой черты, либо одна из заменяющих последовательностей, перечисленных выше.

Комментарий – последовательность символов от символа "процент" (%) до символа "перевод строки", содержащая только печатные символы ASCII или символы горизонтальной табуляции.

Разделитель – один из символов: пробел, вертикальная табуляция, горизонтальная табуляция, возврат каретки, перевод строки, переход на новую страницу либо комментарий. Между любыми двумя лексемами может быть любое число разделителей. Между двумя служебными словами, идентификаторами, изображениями целых и вещественных чисел должен быть по крайней мере один разделитель.

Лексема – последовательность печатных символов ASCII, представляющая служебное слово, идентификатор, изображение, знак операции или символ пунктуации.

Примечание: лексика входного языка Клу-Эльбрус имеет следующие особенности:

1) вместо кодировки ASCII применяется кодировка ДКОИ;

2) управляющие символы не реализованы;

3) вместо одиночной кавычки применяется символ "апостроф";

4) вместо фигурных скобок используются комбинации символов "[*" и "*]";

5) буквы верхнего и нижнего регистров не различаются;

6) разрешены русские буквы в идентификаторах, изображениях строк и комментариях. Введена русская лексика (см. приложение 4).

ПРИЛОЖЕНИЕ 4

РУССКАЯ ЛЕКСИКА И ТЕРМИНОЛОГИЯ ЯЗЫКА КЛУ

П.4.1. Служебные слова

Английская мнемоника	Русская мнемоника	Английская мнемоника	Русская мнемоника
any	*произв*	*if*	*если*
array	*массив*	*in*	*элем*
begin	*начало*	*int*	*цел*
bool	*лог*	*is*	*опер*
break	*прекр*	*iter*	*итер*
cand	*ис*	*itertype*	*итертип*
char	*лит*	*nil*	*нич*
cluster	*кластер*	*null*	*пуст*
continue	*продолж*	*oneof*	*об*
cor	*илис*	*others*	*проч*
cvt	*атд*	*own*	*собств*
do	*цикл*	*proc*	*проц*
down	*конкр*	*proctype*	*процтип*
else	*иначе*	*real*	*вещ*
elseif	*инес*	*record*	*запись*
end	*конец*	*rep*	*пред*
except	*реакц*	*resignal*	*перенос*
exit	*выход*	*return*	*возврат*
false	*ложь*	*returns*	*рез*
for	*для*	*sequence*	*посл*
force	*привед*	*signal*	*ситуация*
has	*содерж*	*signals*	*сит*
string	*строка*	*up*	*абстр*
struct	*структ*	*variant*	*вариант*
tag	*призн*	*when*	*при*
tagcase	*выбор*	*where*	*где*
then	*то*	*while*	*пока*

Английская мнемоника	Русская мнемоника	Английская мнемоника	Русская мнемоника
true	*истина*	*yield*	*выдать*
type	*тип*	*yields*	*выд*

П.4.2. Идентификаторы встроенных операций и ситуаций

Английская мнемоника	Русская мнемоника	Английская мнемоника	Русская мнемоника
equal	*равно*	*min*	*мин*
similar	*подобно*	*from_to_by*	*от_до_шаг*
copy	*копия*	*from_to*	*от_до*
and	*и*	*parse*	*внутр*
or	*или*	*unparse*	*внеш*
not	*не*	*lt*	*мн*
add	*сл*	*le*	*мр*
sub	*вчт*	*ge*	*бр*
mul	*умн*	*gt*	*бл*
div	*дел*	*overflow*	*переполнение*
minus	*минус*	*zero_divide*	*деление_на_нуль*
mod	*мод*	*negative_exponent*	*отрицат_показ*
power	*степ*	*underflow*	*исчезновение*
abs	*абс*	*complex_result*	*комплекс_рез*
max	*макс*	*undefined*	*неопр*
bad_format	*неверный_формат*	*fill_copy*	*заполн_копия*
i2c	*ц2п*	*bottom*	*низ*
c2i	*п2ц*	*top*	*верх*
size	*размер*	*store*	*присв*
empty	*пуст*	*addh*	*вверх*
indexs	*индекс_с*	*addl*	*вниз*
indexc	*индекс_л*	*remh*	*сверху*
c2s	*л2с*	*reml*	*снизу*
concat	*сцепл*	*elements*	*элементы*
append	*присоед*	*indexes*	*индексы*
fetch	*элем*	*similar1*	*подобно1*
rest	*хвост*	*copy1*	*копия1*
substr	*подстр*	*bounds*	*границы*
s2ac	*с2мл*	*negative_size*	*отрицат_размер*
ac2s	*мл2с*	*subseq*	*подпосл*
s2sc	*с2пл*	*replace*	*заменить*
sc2s	*пл2с*	*e2s*	*э2п*
chars	*литеры*	*a2s*	*м2п*
create	*созд*	*s2a*	*п2м*
new	*нов*	*get_f*	*знач_f*
predict	*резерв*	*set_f*	*присв_f*
low	*нигр*	*r_gets_r*	*з_присв_з*

Английская мнемоника	Русская мнемоника	Английская мнемоника	Русская мнемоника
high	*вегр*	*r_gets_s*	*з_присв_с*
set_low	*уст_нигр*	*replace_f*	*заменить_f*
trim	*вырезка*	*s2r*	*с2з*
fill	*заполн*	*r2s*	*з2с*
make_f	*созд_f*	*wrong_tag*	*неверный_признак*
is_f	*есть_f*	*change_f*	*измен_f*
value_f	*значение_f*	*v_gets_v*	*в_присв_в*
o2v	*о2в*	*v_gets_o*	*в_присв_о*
v2o	*в2о*		

П. 4.3. Терминология

В данном пункте приведен рекомендуемый русский вариант терминологии для языка Клу. Даны эквивалентные русские термины для терминов, введенных в авторском описании языка [83].

abstract object – абстрактный объект
abstract type – абстрактный тип
abstract view – абстрактная точка зрения
abstraction – абстракция
accessible object – доступный объект
actual argument – фактический аргумент
argument – аргумент
argument passing – передача аргументов
array type generator – генератор массива
array bounds – границы массива
array constructor – конструктор массива
array state – состояние массива
assignment – присваивание
base type – базовый тип
binary operators – знаки двуместных операций
binding modules – связывание модулей
blank character – пустая литера
block statement – оператор "блок"
body – тело
bool type – тип "логическое"
bool literal – изображение логического
bool operations – операции над логическими
branching – ветвление
break statement – оператор "прекратить"
built-in type (generator) – встроенный тип (генератор типа)
call by sharing – вызов по синониму
compilation environment – контекст компиляции
character – символ
cluster – кластер
coercion – приведение

comment — комментарий
compilation unit — единица компиляции
compile-time constant — статическая константа
compile-time type-checking — статический контроль типов
component — компонента
component name — имя компоненты
component selection — выборка компоненты
component update — изменение компоненты
compound statement — составной оператор
concrete — конкретный
concrete invariant — конкретный вариант
concrete object — конкретный объект
concrete representation — конкретное представление
concrete view — конкретная точка зрения
condition — условие
constant expression — константное выражение
continue statement — оператор "продолжить"
control — управление
control abstraction — абстракция управления
control flow — передача управления
control statements — управляющие конструкции
data abstraction — абстракция данных
data type — тип данных
equated identifier — идентификатор константы
escape sequence — заменяющая последовательность
except statement — оператор реакции
exceptional condition — исключительная ситуация
exceptional termination — завершение (модуля) ситуацией
exceptions — ситуации
exception handling — обработка ситуаций
exit statement — оператор "выход"
exponent — порядок
expression — выражение
external name (reference) — внешнее имя (внешняя ссылка)
field — поле
for statement — оператор "для"
force procedure generator — генератор процедур приведения
formal argument — формальный аргумент
formal parameter — формальный параметр
generator — генератор
get operation — операция выборки
global variable — глобальная переменная
handler — реакция
heading — заголовок
hidden routine — скрытая подпрограмма
identifier — идентификатор
if statement — оператор "если"
immutable object — постоянный объект
implementation of an abstraction — реализация абстракции

indexing – индексация
instantiation – конкретизация
inter-module references – ссылки между модулями
interface specification – описание интерфейса
invocation – вызов
item – элемент
iterator – итератор
library – библиотека
literal – изображение
local variable – локальная переменная
loop statement – оператор цикла
mantissa – мантисса
module – модуль
multiple assignment – групповое присваивание
mutable object – изменяемый объект
name – имя
non-local identifier – нелокальный (глобальный) идентификатор
normal condition – условие нормального завершения
normal termination – нормальное завершение
object – объект
– behavior of – поведение объекта
– creation – создание объекта
– denotation – обозначение объекта
– reference – ссылка на объект
– sharing – синонимы объекта
– state of – состояние объекта
– type of – тип объекта
operation – операция
operator – знак операции
– overloading – полиморфизм операций
order of evaluation – порядок вычисления
others handler – реакция "прочие"
own variable – собственная переменная
parameter – параметр
parametrization – параметризация
parametrized module – параметризованный модуль
primitive operation – первичная операция
printing character – печатный символ
procedure abstraction – процедурная абстракция
procedure – процедура
program – программа
raise an exception – создать ситуацию
record type generator – генератор типов-записей
– component selector – выборка компоненты записи
– constructor – конструктур записей
representation type (**rep**) – тип конкретного представления (*пред*)
reserved word – служебное слово
resignal statement – оператор "перенос"
routine – подпрограмма

scope – область действия
scoping unit – блок
sharing – синоним
signal statement – оператор "ситуация"
signalling of exceptions – создание ситуаций
statement – оператор
syntactic sugar – "синтаксический сахар"
(abbreviations) – (инфиксная форма операций)
tag – признак
tagcase statement – оператор выбора
type – тип
type generator – генератор типов
type specification – обозначение типа
uninitialised variable – неинициализированная переменная
user-defined type – определяемый тип
value – значение
variable – переменная
when handler – реакция "при"
while statement – оператор "пока"
yield statement – оператор "выдать"

ПРИЛОЖЕНИЕ 5

СИНТАКСИС ЯЗЫКА АБВ И ОСОБЕННОСТИ ЯЗЫКА АБВ-ЭЛЬБРУС

П.5.1. Синтаксис языка АБВ

База:

⟨действие⟩ ::= ⟨действие базы⟩ | ⟨действие вычислителя⟩ |
⟨действие анализатора⟩
⟨собственный вид⟩ ::= ⟨вид базы⟩ | ⟨вид вычислителя⟩ |
⟨ вид анализатора⟩
⟨вид базы⟩ ::= ДЕЙСТВ | УКАЗ
⟨синоним⟩ ::= ⟨обозначение⟩ | ⟨изображение действия⟩ |
⟨операция базы⟩ | ⟨последовательное действие⟩
⟨обозначение⟩ ::= ⟨идентификатор⟩ | ⟨составное обозначение⟩ |
⟨косвенное обозначение⟩
⟨идентификатор⟩ ::= ⟨буква⟩ | ⟨идентификатор⟩⟨буква⟩ |
⟨идентификатор⟩⟨цифра⟩
⟨цифра⟩ ::= 0 | ⟨значащая цифра⟩
⟨значащая цифра⟩ ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨составное обозначение⟩ ::=
(АТР⟨простое наименование⟩⟨обозначение⟩)
⟨косвенное обозначение⟩ ::= (ЗНАЧУК⟨обозначение⟩)
⟨наименование⟩ ::=
⟨простое наименование⟩ | н⟨составное обозначение⟩
⟨простое наименование⟩ ::= н⟨идентификатор⟩
⟨изображение действия⟩ ::=
д⟨последовательное действие⟩ | д⟨пустое действие⟩
⟨пустое действие⟩ ::= ( )
⟨действие базы⟩ ::=
⟨простое действие базы⟩ | ⟨составное действие базы⟩
⟨составное действие базы⟩ ::=
⟨последовательное действие⟩ | ⟨параллельное действие⟩
⟨альтернативное действие⟩
⟨последовательное действие⟩ ::= (ПОСЛ⟨список этапов⟩)
⟨список этапов⟩ ::= ⟨этап⟩ | ⟨список этапов⟩⟨этап⟩
⟨этап⟩ ::= ⟨оператор базы⟩ | ⟨оператор анализатора⟩

⟨ составное действие базы ⟩ | ⟨ составное действие вычислителя ⟩ | ⟨ операция прерывание ⟩ | ⟨ операция вывод ⟩
⟨ программа ⟩ ::= ⟨ последовательное действие ⟩
⟨ параллельное действие ⟩ ::= (ПАР ⟨ список ветвей ⟩)
⟨ список ветвей ⟩ ::= ⟨ ветвь ⟩ | ⟨ список ветвей ⟩ ⟨ ветвь ⟩
⟨ ветвь ⟩ ::= ⟨ последовательное действие ⟩
⟨ альтернативное действие ⟩ ::=
(АЛЬТ ⟨ условие альтернативного действия ⟩
⟨ элемент альтернативного действия ⟩
⟨ элемент альтернативного действия ⟩)
⟨ условие альтернативного действия ⟩ ::=
⟨ обозначение ⟩ | ⟨ операция объект ⟩ | ⟨ операция сравнение строк ⟩
| ⟨ операция сравнение действий ⟩ | ⟨ операция сравнение имен ⟩ |
⟨ операция преобразование строки в логическое ⟩
⟨ элемент альтернативного действия ⟩ ::=
⟨ оператор базы ⟩ | ⟨ оператор анализатора ⟩ |
⟨ составное действие базы ⟩ | ⟨ составное действие вычислителя ⟩
| ⟨ операция прерывание ⟩ | ⟨ операция вывод ⟩ | ⟨ пустое действие ⟩
⟨ простое действие базы ⟩ ::=
⟨ оператор базы ⟩ | ⟨ операция базы ⟩
⟨ оператор базы ⟩ ::=
⟨ оператор ввести синоним ⟩ |
⟨ оператор ликвидировать синоним ⟩ |
⟨ оператор присвоить ⟩ | ⟨ оператор возобновить ⟩ |
⟨ оператор завершить ⟩ | ⟨ оператор освободить память ⟩
⟨ операция базы ⟩ ::=
⟨ операция прерывание ⟩ | ⟨ операция преобразование строки ⟩ |
⟨ операция взятие вида ⟩ | ⟨ операция объект ⟩ |
⟨ операция сравнение действий ⟩ | ⟨ операция сравнение строк ⟩ |
⟨ операция сравнение имен ⟩ | ⟨ операция память ⟩ |
⟨ операция указатель ⟩
⟨ оператор ввести синоним ⟩ ::=
(ВСИН ⟨ обозначение ⟩ ⟨ простое наименование ⟩)
⟨ оператор присвоить ⟩ ::=
(ПРИСВ ⟨ синоним ⟩ ⟨ обозначение ⟩)
⟨ оператор ликвидировать синоним ⟩ ::=
(ЛИКВ ⟨ наименование ⟩)
⟨ оператор освободить память ⟩ ::=
(ОСВПАМ ⟨ обозначение ⟩)
⟨ оператор завершить ⟩ ::= (ЗАВЕРШ) | (ЗАВЕРШ ⟨ синоним ⟩)
⟨ оператор возобновить ⟩ ::= (ВОЗОБН)
⟨ операция прерывание ⟩ ::= (ПРЕР ⟨ обозначение ⟩)
⟨ операция взятие вида ⟩ ::= (ВИД ⟨ обозначение ⟩)
⟨ операция сравнение имен ⟩ ::=
(⟨ знак операции сравнение имен ⟩ ⟨ синоним ⟩ ⟨ синоним ⟩)
⟨ знак операции сравнение имен ⟩ ::= РАВИМ | НРАВИМ
⟨ операция сравнение действий ⟩ ::=
(⟨ знак операции сравнение действий ⟩
⟨ операнд операции сравнение действий⟩

⟨ операнд операции сравнение действий ⟩)
⟨ знак операции сравнение действий ⟩ ::= РАВД | НРАВД
⟨ операнд операции сравнение действий ⟩ ::=
⟨ обозначение ⟩ | ⟨ операция взятие вида ⟩
⟨ операция сравнение строк ⟩ ::=
(⟨ знак операции сравнение строк ⟩
⟨ операнд операции сравнение строк ⟩
⟨ операнд операции сравнение строк ⟩)
⟨ знак операции сравнение строк ⟩ ::= РАВС | НРАВС
⟨ операнд операции сравнение строк ⟩ ::=
⟨ обозначение ⟩ | ⟨ операция объект ⟩
⟨ операция преобразования строки ⟩ ::=
⟨ операция преобразование строки в целое единичной длины ⟩ |
⟨ операция преобразование строки в целое ⟩ |
⟨ операция преобразование строки в приближенное ⟩ |
⟨ операция преобразование строки в шкалу ⟩ |
⟨ операция преобразование строки в логическое ⟩ |
⟨ операция преобразование строки в действие ⟩ |
⟨ операция преобразование строки в формат ⟩
⟨ операция преобразование строки в целое единичной длины ⟩ ::=
(ПСЦ1 ⟨ операнд операции над строками ⟩)
⟨ операция преобразование строки в целое ⟩ ::=
(ПСЦ ⟨ операнд операции над строками ⟩
⟨ генератор целых единичной длины ⟩)
⟨ операция преобразование строки в приближенное ⟩ ::=
(ПСП ⟨ операнд операции над строками ⟩
⟨ генератор целых единичной длины ⟩)
⟨ операция преобразование строки в логическое ⟩ ::=
(ПСЛОГ ⟨ операнд операции над строками ⟩)
⟨ операция преобразование строки в действие ⟩ ::=
(ПСД ⟨ операнд операции над строками ⟩)
⟨ операция преобразование строки в формат ⟩ ::=
(ПСФ ⟨ операнд операции над строками ⟩)
⟨ операция память ⟩ ::=
(ПАМ ⟨ генератор целых единичной длины ⟩
⟨ генератор целых единичной длины ⟩)
⟨ операция указатель ⟩ ::= (ОБУК ⟨ синоним ⟩)
⟨ операция объект ⟩ ::=
(ОБЦ1 ⟨ генератор целых единичной длины ⟩)
(ОБЦ ⟨ генератор целых ⟩)
(ОБПР ⟨ генератор приближенных ⟩)
(ОБШ ⟨ генератор шкал ⟩)
(ОБЛОГ ⟨ генератор логических ⟩)
(ОБСТР ⟨ операнд операции над строками ⟩)
(ОБФОР ⟨ формат ⟩)

Вычислитель:

⟨ вид вычислителя ⟩ ::=
ЦЕЛ | ЦЕЛ1 | ПРИБ | ШКАЛА | ЛОГ | ПОЛЕ

⟨изображение целого⟩ ::=
ц⟨изображение длины⟩ ц⟨целое⟩
⟨целое⟩ ::= ⟨целое без знака⟩ | + ⟨целое без знака⟩ |
– ⟨целое без знака⟩
⟨целое без знака⟩ ::= ⟨цифра⟩ | ⟨целое без знака⟩⟨цифра⟩
⟨изображение длины⟩ ::=
⟨значащая цифра⟩ | ⟨изображение длины⟩⟨значащая цифра⟩
⟨изображение целого единичной длины⟩ ::= цц ⟨целое⟩
⟨изображение приближенного⟩ ::=
п⟨изображение длины⟩ п⟨приближенное⟩
⟨приближенное⟩ ::=
⟨приближенное без знака⟩ | + ⟨приближенное без знака⟩ |
– ⟨приближенное без знака⟩
⟨приближенное без знака⟩ ::=
⟨десятичное⟩ | ⟨порядок⟩ | ⟨десятичное⟩ ⟨порядок⟩
⟨десятичное⟩ ::= ⟨целое без знака⟩ | ⟨правильная дробь⟩ |
⟨целое без знака⟩ ⟨правильная дробь⟩
⟨правильная дробь⟩ ::= .⟨целое без знака⟩
⟨порядок⟩ ::= $_{10}$⟨целое⟩
⟨изображение логического⟩ ::= л⟨двоичная цифра⟩
⟨двоичная цифра⟩ ::= 0 | 1
⟨изображение шкалы⟩ ::=
ш⟨изображение длины⟩ ш⟨шкала⟩
⟨шкала⟩ ::= ⟨двоичная цифра⟩ | ⟨шкала⟩⟨двоичная цифра⟩
⟨действие вычислителя⟩ ::=
⟨простое действие вычислителя⟩ |
⟨составное действие вычислителя⟩
⟨простое действие вычислителя⟩ ::=
⟨операция вычислителя⟩ | ⟨оператор вычислителя⟩
⟨операция вычислителя⟩ ::=
⟨генератор целых⟩ | ⟨генератор целых единичной длины⟩ |
⟨генератор приближенных⟩ | ⟨генератор шкал⟩ |
⟨генератор логических⟩
⟨генератор целых⟩ ::=
⟨двуместная операция над целыми⟩ |
⟨одноместная операция над целыми⟩ |
⟨операция преобразование в целое⟩ |
⟨операция чтение целого⟩ | ⟨операция запись целого⟩ |
⟨значение-целое⟩ | ⟨изображение целого⟩
⟨двуместная операция над целыми⟩ ::=
(⟨знак двуместной операции над целыми⟩
⟨генератор целых⟩ ⟨генератор целых⟩)
⟨знак двуместной операции над целыми⟩ ::=
ЦС | ЦСУ | ЦВ | ЦВУ | ЦУ | ЦУУ | ЦСТ
⟨одноместная операция над целыми⟩ ::=
(⟨знак одноместной операции над целыми⟩
⟨генератор целых⟩)
⟨знак одноместной операции над целыми⟩ ::= ЦУД | ЦУК

```
⟨операция преобразование в целое⟩ ::=
        (ПЦ1Ц ⟨генератор целых единичной длины⟩) |
        (ПШЦ  ⟨генератор шкал⟩) |
        (ОКР  ⟨генератор приближенных⟩) |
        (ЦЧ   ⟨генератор приближенных⟩) |
⟨генератор целых единичной длины⟩ ::=
        ⟨двуместная операция над целыми единичной длины⟩ |
        ⟨операция длина⟩ | ⟨операция длина строки⟩ |
        ⟨операция длина поля⟩ | ⟨операция уровеиь⟩ |
        ⟨операция преобразование строки в целое единичной длины⟩ |
        ⟨операция запись целого единичной длины⟩ |
        ⟨значение-целое единичной длины⟩ |
        ⟨изображение целого единичной длины⟩
⟨двуместная операция над целыми единичной длины⟩ ::=
        (⟨знак двуместной операции над целыми единичной длины⟩
         ⟨генератор целых единичной длины⟩
         ⟨генератор целых единичной длины⟩)
⟨знак двуместной операции над целыми единичной длины⟩ ::=
        Ц1С | Ц1В | Ц1У | Ц1СТ
⟨операция длина⟩ ::= (ДЛ ⟨операнд операции длина⟩)
⟨операнд операции длина⟩ ::=
        ⟨генератор целых⟩ | ⟨генератор приближенных⟩ |
        ⟨генератор шкал⟩
⟨операция длина поля⟩ ::= (ДЛПОЛЕ ⟨значение-поле⟩)
⟨операция уровень⟩ ::= (УРОВ ⟨значение-поле⟩)
⟨операция преобразование в целое единичной длины⟩ ::=
        (ПЦЦ1 ⟨генератор целых⟩) | (ПШЦ1 ⟨генератор шкал⟩)
⟨операция порядок⟩ ::= (ПОР ⟨генератор приближенных⟩)
⟨генератор приближенных⟩ ::=
        ⟨двуместная операция над приближенными⟩ |
        ⟨одноместная операция над приближенными⟩ |
        ⟨операция приближенного деления целых⟩ |
        ⟨операция возведение приближенного в степень⟩ |
        ⟨операция преобразование в приближенное⟩ |
        ⟨операция чтение приближенного⟩ |
        ⟨операция запись приближенного⟩ |
        ⟨значение-приближенное⟩ | ⟨изображение-приближенного⟩
⟨двуместная операция над приближенными⟩ ::=
        (⟨знак двуместной операции над приближенными⟩
        ⟨генератор приближенных⟩ ⟨генератор приближенных⟩)
⟨знак двуместной операции над приближенными⟩ ::= ПС | ПВ | ПУ | ПД
⟨одноместная операция над приближенными⟩ ::=
        (⟨знак одноместной операции над приближенными⟩
         ⟨генератор приближенных⟩)
⟨знак одноместной операции над приближенными⟩ ::=
        ПУК | ПУД
⟨операция приближенное деление целых⟩ ::=
        (ПРДЦ ⟨генератор целых⟩ ⟨генератор целых⟩)
```

⟨ операция возведение приближенного в степень ⟩ ::=
(ПСТ ⟨ генератор приближенных ⟩
⟨ генератор целых единичной длины ⟩)
⟨ операция преобразование в приближенное ⟩ ::=
(ПЦ1П ⟨ генератор целых единичной длины ⟩)
(ПЦП ⟨ генератор целых ⟩) | (ПШМ ⟨ генератор шкал ⟩)
⟨ генератор шкал ⟩ ::=
⟨ двуместная операция над шкалами ⟩ |
⟨ одноместная операция над шкалами ⟩ |
⟨ операция сдвиг ⟩ | ⟨ операция преобразование в шкалу ⟩ |
⟨ операция чтение шкалы ⟩ | ⟨ операция запись шкалы ⟩ |
⟨ значение-шкала ⟩ | ⟨ изображение шкалы ⟩
⟨ двуместная операция над шкалами ⟩ ::=
(⟨ знак двуместной операции над шкалами ⟩
⟨ генератор шкал ⟩ ⟨ генератор шкал ⟩)
⟨ знак двуместной операции над шкалами ⟩ ::= КОН | ДИЗ | НЭКВ
⟨ одноместная операция над шкалами ⟩ ::=
(⟨ знак одноместной операции над шкалами ⟩ ⟨ генератор шкал ⟩)
⟨ знак одноместной операции над шкалами ⟩ ::= ОТР | УДШ | УДШ̆
⟨ операция сдвиг ⟩ ::=
(⟨ знак операции сдвиг ⟩ ⟨ генератор шкал ⟩
⟨ генератор целых единичной длины ⟩)
⟨ знак операции сдвиг ⟩ ::= СДП | СДЛ
⟨ операция преобразование в шкалу ⟩ ::=
(ПЦ1Ш ⟨ генератор целых единичной длины ⟩)
(ПЦШ ⟨ генератор целых ⟩)
(ПМШ ⟨ генератор приближенных ⟩)
⟨ генератор логических ⟩ ::=
⟨ двуместная логическая операция ⟩ | ⟨ операция отношение ⟩ |
⟨ операция отрицание ⟩ | ⟨ операция чтение логического ⟩ |
⟨ операция запись логического ⟩ | ⟨ значение-логическое ⟩ |
⟨ изображение логического ⟩
⟨ двуместная логическая операция ⟩ ::=
(⟨ знак двуместной логической операции ⟩
⟨ генератор логических ⟩ ⟨ генератор логических ⟩)
⟨ знак двуместной логической операции ⟩ ::= КОНЛ | ДИЗЛ | ИМПЛ
⟨ операция отрицание ⟩ ::= (ОТРЛ ⟨ генератор логических ⟩)
⟨ операция отношение ⟩ ::=
⟨ операция отношение целых единичной длины ⟩ |
⟨ операция отношение целых ⟩ |
⟨ операция отношение приближенных ⟩ |
⟨ операция отношение шкал ⟩ |
⟨ операция отношение логических ⟩
⟨ операция отношение целых единичной длины ⟩ ::=
(⟨ знак операции отношение целых единичной длины ⟩
⟨ генератор целых единичной длины ⟩
⟨ генератор целых единичной длины ⟩)
⟨ знак операции отношение целых единичной длины ⟩ ::=
РАВЦ1 | НРАВЦ1 | МЕНЦ1 | МРАВЦ1 | БОЛЦ1 | БОЛЦ1 | БРАВЦ1

⟨ операция отношение целых ⟩ ::=
(⟨ знак операции отношение целых ⟩
⟨ генератор целых ⟩ ⟨ генератор целых ⟩)
⟨ знак операции отношение целых ⟩ ::=
РАВЦ | НРАВЦ | МЕНЦ | МРАВЦ | БОЛЦ | БРАВЦ
⟨ операция отношение приближенных ⟩ ::=
(⟨ знак операции отношение приближенных ⟩
⟨ генератор приближенных ⟩ ⟨ генератор приближенных ⟩)
⟨ знак операции отношение приближенных ⟩ ::=
РАВПР | НРАВПР | МЕНПР | МРАВПР | БОЛПР | БРАВПР
⟨ операция отношение шкал ⟩ ::=
(⟨ знак операции отношение шкал ⟩
⟨ генератор шкал ⟩ ⟨ генератор шкал ⟩)
⟨ знак операции отношение шкал ⟩ ::= РАВШ | НРАВШ
⟨ операция отношение логических ⟩ ::=
(⟨ знак операции отношение логических ⟩
⟨ генератор логических ⟩ ⟨ генератор логических ⟩)
⟨ знак операции отношение логических ⟩ ::= РАВЛ | НРАВЛ
⟨ операция значение ⟩ ::=
⟨ значение-целое единичной длины ⟩ | ⟨ значение-целое ⟩ |
⟨ значение-приближенное ⟩ | ⟨ значение-шкала ⟩ |
⟨ значение-логическое ⟩ | ⟨ значение-поле ⟩ | ⟨ значение-строка ⟩ |
⟨ значение-формат ⟩
⟨ значение-целое единичной длины ⟩ ::=
(ЗНЦ1 ⟨ обозначение ⟩) |
(ЗНЦ1 ⟨ операция преобразование строки в целое
единичной длины ⟩)
⟨ значение-целое ⟩ ::=
(ЗНЦ ⟨ обозначение ⟩) |
(ЗНЦ ⟨ операция преобразование строки в целое ⟩)
⟨ значение-приближенное ⟩ ::=
(ЗНПР ⟨ обозначение ⟩) |
(ЗНПР ⟨ операция преобразование строки в приближенное ⟩)
⟨ значение-шкала ⟩ ::=
(ЗНШ ⟨ обозначение ⟩) |
(ЗНШ ⟨ операция преобразование строки в шкалу ⟩)
⟨ значение-логическое ⟩ ::=
(ЗНЛ ⟨ обозначение ⟩) |
(ЗНЛ ⟨ операция преобразование строки в логическое ⟩)
⟨ значение-поле ⟩ ::= (ЗНПОЛЕ ⟨ обозначение ⟩)
⟨ операция запись ⟩ ::=
⟨ операция запись целого единичной длины ⟩ |
⟨ операция запись целого ⟩ |
⟨ операция запись приближенного ⟩ |
⟨ операция запись шкалы ⟩ |
⟨ операция запись логического ⟩ |
⟨ операция запись строки ⟩
⟨ операция запись целого единичной длины ⟩ ::=
(ЗАПЦ1 ⟨ ячейка ⟩ ⟨ генератор целых единичной длины ⟩)

```
⟨операция запись целого⟩ ::=
        (ЗАПЦ ⟨часть поля⟩ ⟨генератор целых⟩)
⟨операция запись приближенного⟩ ::=
        (ЗАППР ⟨часть поля⟩ ⟨генератор приближенных⟩)
⟨операция запись шкалы⟩ ::=
        (ЗАПШ ⟨часть поля⟩ ⟨генератор шкал⟩)
⟨операция запись логического⟩ ::=
        (ЗАПЛ ⟨ячейка⟩ ⟨генератор логических⟩)
⟨операция запись строки⟩ ::=
        (ЗАПСТ ⟨часть поля⟩ ⟨операнд операции над строками⟩)
⟨ячейка⟩ ::= ⟨значение-поле⟩ ⟨генератор целых единичной длины⟩
⟨часть поля⟩ ::= ⟨ячейка⟩ ⟨генератор целых единичной длины⟩
⟨операция чтение⟩ ::=
        ⟨операция чтение целого единичной длины⟩ |
        ⟨операция чтение целого⟩ | ⟨операция чтение приближенного⟩ |
        ⟨операция чтение шкалы⟩ | ⟨операция чтение логического⟩ |
        ⟨операция чтение строки⟩
⟨операция чтение целого единичной длины⟩ ::= (ЧИТЦ1 ⟨ячейка⟩)
⟨операция чтение целого⟩ ::= (ЧИТЦ ⟨часть поля⟩)
⟨операция чтение приближенного⟩ ::= (ЧИТПР ⟨часть поля⟩)
⟨операция чтение логического⟩ ::= (ЧИТЛ ⟨ячейка⟩)
⟨операция чтение строки⟩ ::= (ЧИТСТ ⟨часть поля⟩)
⟨оператор вычислителя⟩ ::=
        ⟨оператор переслать⟩ | ⟨оператор деление целых⟩ |
        ⟨оператор перехода⟩
⟨оператор переслать⟩ ::= (ПЕРЕС ⟨часть поля⟩ ⟨ячейка⟩)
⟨оператор деление целых⟩ ::= (ДЕЛЦ ⟨генератор целых⟩
                                   ⟨генератор целых⟩)
⟨составное действие вычислителя⟩ ::=
        (СОСВ ⟨список элементов составного действия вычислителя⟩)
⟨список элементов составного действия вычислителя⟩
        ::= ⟨элемент составного действия вычислителя⟩ |
        ⟨список элементов составного действия вычислителя⟩
        ⟨элемент составного действия вычислителя⟩
⟨элемент составного действия вычислителя⟩ ::=
        ⟨оператор вычислителя⟩ | ⟨операция запись⟩ | ⟨метка⟩
⟨метка⟩ ::= ⟨идентификатор⟩
⟨оператор перехода⟩ ::=
        ⟨оператор безусловного перехода⟩ |
        ⟨оператор условного перехода⟩
⟨оператор безусловного перехода⟩ ::= (БП ⟨метка⟩)
⟨оператор условного перехода⟩ ::= (УП ⟨условие⟩ ⟨метка⟩)
⟨условие⟩ ::= ⟨операнд логической операции⟩
```

Анализатор:

```
⟨вид анализатора⟩ ::= СТРОКА | ФОРМАТ
⟨строка⟩ ::≐ '⟨список элементов строки⟩'
⟨список элементов строки⟩ ::=
        ⟨элемент строки⟩ | ⟨список элементов строки⟩ | ⟨элемент строки⟩
```

⟨ элемент строки ⟩ ::= ⟨ символ строки ⟩ | ⟨ строка ⟩
⟨ символ строки ⟩ ::= ⟨ основной символ ⟩ | ⟨ прочий символ ⟩
⟨ основной символ ⟩ ::= ⟨ буква ⟩ | ⟨ цифра ⟩ | ⟨ ограничитель ⟩
⟨ ограничитель ⟩ ::= (|) | | ‘ | ’ | $_{10}$ | + | — | . | _
⟨ операция длина строки ⟩ ::=

(⟨ знак операции длина строки ⟩ ⟨ операнд операции над строками ⟩)

⟨ знак операции длина строки ⟩ ::= ЧЭЛ | ЧСИМ
⟨ изображение формата ⟩ ::= ф ⟨ тело формата ⟩ ф
⟨ тело формата ⟩ ::=

⟨ формат шкалы ⟩ | ⟨ формат целого ⟩ | ⟨ формат приближенного ⟩ |
⟨ формат текста ⟩

⟨ формат шкалы ⟩ ::= ⟨ повторитель ⟩ В
⟨ формат целого ⟩ ::= З ⟨ повторитель ⟩ С
⟨ формат приближенного ⟩ ::=

З ⟨ повторитель0 ⟩ С.⟨ повторитель ⟩ $С_{10}$З ⟨ повторитель ⟩ С |
З ⟨ повторитель0 ⟩ С.⟨ повторитель ⟩ С

⟨ повторитель ⟩ ::= ⟨ значащая цифра ⟩ | ⟨ повторитель ⟩ ⟨ цифра ⟩
⟨ повторитель0 ⟩ := 0 | ⟨ повторитель ⟩
⟨ действие анализатора ⟩ ::=

⟨ генератор строк ⟩ | ⟨ оператор анализатора ⟩ |
⟨ значение-формат ⟩

⟨ значение-формат ⟩ ::=

(ЗНФ ⟨ операнд операции значение-формат ⟩)

⟨ операнд операции значение-формат ⟩ ::=

⟨ обозначение ⟩ | ⟨ операция преобразование строки в формат ⟩

⟨ генератор строк ⟩ ::=

⟨ операция сцепление ⟩ | ⟨ операция взять в кавычки ⟩ |
⟨ операция снять кавычки ⟩ | ⟨ операция чтение строки ⟩ |
⟨ операция запись строки ⟩ |
⟨ операция преобразование в строку ⟩ |
⟨ операция ввод ⟩ | ⟨ операция вывод ⟩ | ⟨ операция копия ⟩ |
⟨ значение-строка ⟩

⟨ операция сцепление ⟩ ::=

(СЦЕПЛ ⟨ операнд операции над строками ⟩
⟨ операнд операции над строками ⟩)

⟨ операции взять в кавычки ⟩ ::=

(КАВЫЧ ⟨ операнд операции над строками ⟩)

⟨ операция снять кавычки ⟩ ::=

(СНКАВ ⟨ операнд операции над строками ⟩)

⟨ операция копия ⟩ ::=

(КОПИЯ ⟨ операнд операции над строками ⟩)

⟨ операнд операции над строками ⟩ ::= ⟨ строка ⟩ | ⟨ генератор строк ⟩
⟨ значение-строка ⟩ ::= (ЗНСТР ⟨ обозначение ⟩)
⟨ операция преобразование в строку ⟩ ::=

⟨ операция преобразования шкалы в строку ⟩ |
⟨ операция преобразование целого единичной длины в строку ⟩ |
⟨ операция преобразование целого в строку ⟩ |
⟨ операция преобразование приближенного в строку ⟩ |

⟨ операция преобразование логического в строку ⟩
⟨ операция преобразование шкалы в строку ⟩ ::=
(ПШС ⟨ генератор шкал ⟩ ⟨ формат ⟩)
⟨ операция преобразования целого единичной длины в строку ⟩ ::=
(ПЦ1С ⟨ генератор целых единичной длины ⟩ ⟨ формат ⟩)
⟨ операция преобразование целого в строку ⟩ ::=
(ПЦС ⟨ генератор целых ⟩ ⟨ формат ⟩)
⟨ операция преобразование приближенного в строку ⟩ ::=
(ППРС ⟨ генератор приближенных ⟩ ⟨ формат ⟩)
⟨ операция преобразование логического в строку ⟩ ::=
(ПЛЮГС ⟨ генератор логических ⟩)
⟨ формат ⟩ ::= ⟨ изображение формата ⟩ | ⟨ значение-формат ⟩
⟨ операция вывод ⟩ ::=
(ВЫВОД ⟨ операнд операции над строками ⟩ ⟨ формат ⟩)
⟨ формат текста ⟩ ::= ⟨ список элементов формата текста ⟩
⟨ список элементов формата текста ⟩ ::=
⟨ элемент формата текста ⟩ |
⟨ список элементов формата текста ⟩ ⟨ элемент формата текста ⟩
⟨ элемент формата текста ⟩ ::=
⟨ управляющий символ ⟩ | ⟨ повторитель ⟩ ⟨ управляющий символ ⟩
⟨ управляющий символ ⟩ ::= S | / | П | % | X
⟨ операция ввод ⟩ ::= (ВВОД)
⟨ оператор анализатора ⟩ ::=
⟨ оператор найти вхождение ⟩ | ⟨ оператор выделить ⟩
⟨ оператор найти вхождение ⟩ ::=
(ВХОЖД ⟨ операнд операции над строками ⟩
⟨ операнд операции над строками ⟩)
⟨ оператор выделить ⟩ ::=
(ВЫД ⟨ генератор целых единичной длины ⟩
⟨ операнд операции над строками ⟩)

П.5.2. Особенности языка АБВ-Эльбрус

1. В изображениях строк вместо парных кавычек ‘’ используются заменяющие последовательности (′и′).

2. Вместо символа $1_{ло}$ в изображениях чисел и форматов используется символ "Е".

3. Во входной язык введен дополнительный собственный вид вычислителя ПРИБ1 (приближенное единичной длины), значения которого представляются числами формата ф64. Введены также изображение приближенного единичной длины и соответствующие операции. В связи с этим приведенные синтаксические правила изменяются и дополняются следующим образом:

⟨ операция преобразование строки ⟩ ::= ... |
⟨ операция преобразование строки в приближенное единичной длины ⟩
⟨ операция преобразование строки в приближенное единичюй длины ⟩ ::=
(ПСП1 ⟨ операнд операции над строками ⟩)

⟨ операция объект ⟩ ::= . . . |
(ОБПР1 ⟨ генератор приближенных единичной длины ⟩)
⟨ вид вычислителя ⟩ ::= . . . | ПРИБ1
⟨ изображение приближенного единичной длины ⟩ ::=
пп ⟨ приближенное ⟩
⟨ операция вычислителя ⟩ ::= . . . |
⟨ генератор приближенных единичной длины ⟩
⟨ генератор приближенных единичной длины ⟩ ::=
⟨ двуместная операция над приближенными единичной длины ⟩ |
⟨ операция преобразование в приближенное единичной длины ⟩ |
⟨ операция чтение приближенного единичной длины ⟩ |
⟨ операция чтение приближенного единичной длины ⟩ |
⟨ значение-приближенное единичной длины ⟩ |
⟨ изображение приближенного единичной длины ⟩
⟨ двуместная операция над приближенными единичной длины ⟩ ::=
(⟨ знак двуместной операции над приближенными единичной длины ⟩ ⟨ генератор приближенных единичной длины ⟩
⟨ генератор приближенных единичной длины ⟩)
⟨ знак двуместной операции над приближенными единичной длины ⟩ ::=
П1С | П1В | П1Д | П1СТ
⟨ операция преобразование в приближенное единичной длины ⟩ ::=
(ПЦ1П1 ⟨ генератор целых единичной длины ⟩) |
(ППРП1 ⟨ генератор приближенных ⟩)
⟨ операция отношение ⟩ ::= . . . |
⟨ операция отношение приближенных единичной длины ⟩
⟨ операция отношение приближенных единичной длины ⟩ ::=
(⟨ знак операции отношения приближенных единичной длины ⟩
⟨ генератор приближенных единичной длины ⟩
⟨ генератор приближенных единичной длины ⟩)
⟨ знак операции отношение приближенных единичной длины ⟩ ::=
РАВПР1 | НРАВПР1 | МЕНПР1 | МРАВПР1 | БОЛПР1 | БРАВПР1
⟨ операция значение ⟩ ::= . . . |
⟨ значение-приближенное единичной длины ⟩
значение-приближенное единичной длины ::=
(ЗНПР1 ⟨ обозначение ⟩) |
(ЗНПР1 ⟨ операция преобразование строки в приближенное единичной длины ⟩)
⟨ операция запись ⟩ ::= . . . |
⟨ операция запись приближенного единичной длины ⟩
⟨ операция запись приближенного единичной длины ⟩ ::=
(ЗАППР1 ⟨ ячейка ⟩ ⟨ генератор приближенных единичной длины ⟩
)
⟨ операция чтение ⟩ ::= . . . |
⟨ операция чтение приближенного единичной длины ⟩
⟨ операция чтение приближенного единичной длины ⟩ ::=
(ЧИТПР1 ⟨ ячейка ⟩)
⟨ операция преобразование в строку ⟩ ::= . . .
⟨ операция преобразование приближенного единичной длины в строку ⟩

<операция преобразования приближенного единичной длины в строку>
::= (ПП1С <генератор приближенных единичной длины> <формат>)

Стандартные прерывания и идентификаторы реакций на них. При выполнении простых действий языка АБВ могут возникать прерывания. Каждое из этих прерываний имеет в языке АБВ свой идентификатор, что позволяет с помощью оператора присваивания переопределить реакцию на данное прерывание.

Прерывания в базе:

РАВДПРЕР – операнд операции *сравнение действий* не ссылается на объект вида ДЕЙСТВ;

РАВСПРЕР – операнд операции *сравнение строк* не ссылается на объект вида СТРОКА;

ОСВПРЕР – операнд оператора *освободить память* не ссылается на объект вида ПОЛЕ;

СИНТПРЕР – в операции *преобразование строки* строка не соответствует синтаксису значения;

ВСИНПРЕР – в операторе *ввести синоним* либо первый операнд не ссылается на объект вида ДЕЙСТВ, либо его значение не определено;

ЛИКВПРЕР – в операторе *ликвидировать синоним* ликвидируется несуществующий синоним;

ИДПРЕР – идентификатор не ссылается ни на какой объект;

АТРПРЕР – составное обозначение не ссылается ни на какой объект;

УКАЗПРЕР – косвенное обозначение не ссылается ни на какой объект;

ВЗБНПРЕР – оператор *возобновить* выполняется вне реакции на прерывание;

ПРЕРПРЕР – в операции *прерывание* реакция либо не определена, либо запускается вне своей области определения;

ПРСВПРЕР – операнды оператора *присвоить* ссылаются на объекты различных видов;

ПАМПРЕР – неверно задан операнд операции *память* (уровень или размер поля);

ДЛПСПРЕР – неверно задана длина в операциях *преобразование* строки либо длина значения превышает заданную длину;

Прерывания в вычислителе:

ПЕРЕПЦ1 – переполнение в операциях над целыми единичной длины;

ПЕРЕПЦЕЛ – переполнение в операциях над целыми;

ПЕРЕППР – переполнение в операциях над приближенными;

ПЕРЕППР1 – переполнение в операциях над приближенными единичной длины;

ДЛИНПРЕР – в операциях над длинными значениями длины операндов различны;

ДЕЛНОЛЬ – деление на нуль;

SQRTПРЕР – операнд операции *квадратный корень* отрицателен;

LNПРЕР – операнд операции *логарифм* неположителен;

TGПРЕР – операнд операции *тангенс* близок к $\frac{\pi}{2} + k\pi$; $k = 0, \pm 1, \pm 2 \dots$;

ARCSПРЕР – операнд операции *арксинус* выходит за пределы отрезка $[-1, 1]$;

ЗНПРЕР – в операциях *значение* вид операнда не соответствует знаку операции;

ШКАЛПРЕР – в операциях *преобразование шкалы в целое, в целое единичной длины* или *приближенное* шкала не является стандартным представлением значения соответствующего вида:

УКПРЕР – укорачивается значение длины 1;

ЦУКПРЕР – укорачивается целое число, которое не может быть представлено как число с меньшей длиной;

ПРЦ1ПРЕР – длина значения, преобразуемое в целое единичной длины, превосходит единицу;

ПОЛЕПРЕР – поле-операнд операции *запись* или *чтение* не находится в оперативной памяти;

ГРАНПРЕР – часть поля выходит за пределы поля либо пересылаемая часть поля превосходит часть поля, куда делается пересылка;

СДВПРЕР – операнд операции сдвиг – отрицательное число;

СТЕППРЕР – показатель степени в операциях возведения в степень отрицателен;

Прерывание в анализаторе:

СНКВПРЕР – результат операции *снять кавычки* не является строкой;

СТРЧПРЕР – отсутствует переход на новую строчку в формате при выводе;

СТРНПРЕР – отсутствует переход на новую страницу в формате при выводе;

ФОРМПРЕР – в операции *преобразование в строку* формат не соответствует знаку операции.

СПИСОК ЛИТЕРАТУРЫ

1. *Агафонов В.Н.* Языки и средства спецификации программ // Требования и спецификации в разработке программ. – М.: Мир, 1984. – С. 285–344.
2. *Агафонов В.Н.* Типы и абстракции данных в языках программирования // Данные в языках программирования – М.: Мир, 1982. – С. 269–327.
3. Алгол-68. Методы реализации / Под ред. Г.С. Цейтина. – Л.: Изд-во ЛГУ, 1976. – 224 с.
4. Алгоритмический язык Алгол-60. Модифицированное сообщение. – М.: Мир, 1982. – 71 с.
5. *Андреев В.М., Голосов И.С., Самойленко С.М.* Выборочная оптимизация в трансляторе Фортран-Эльбрус // Трансляция и оптимизация программ. – Новосибирск: ВЦ СО АН СССР, 1983. – С. 51–64.
6. Базисный Рефал и его реализация на вычислительных машинах. – М.: ЦНИПИАСС, 1977. – 258 с.
7. *Бабаев И.О., Новиков Ф.А., Петрушина Т.И.* Язык Декарт – входной язык системы СПОРА // Прикладная информатика; Вып. 1. – М.: Финансы и статистика, 1981. – С. 35–73.
8. *Бабаян Б.А., Волконский В.Ю., Пентковский В.М.* Системная поддержка модульного программирования. – Препринт № 11/ИТМ и ВТ АН СССР. – М., 1985. – 70 с.
9. *Бабаян Б.А., Сахин Ю.Х.* Система Эльбрус. – Программирование. – 1980. – № 6. – С. 72–86.
10. *Бабаян Б.А., Пентковский В.М.* Языковая модель системной поддержки модульного программирования. – Препринт № 7/ИТМ и ВТ АН СССР. – М., 1985. – 59 с.
11. *Баурн С.* Операционная система UNIX. – М.: Мир, 1986. – 463 с.
12. *Болдинова Н.Н., Смертин А.Н.* Методы управления структурами данных и памятью в трансляторе Снобол-Эльбрус// Программирование, 1985. – № 1. – С. 44–49.
13. *Броль В.В., Гущин В.М., Яковлев В.Б.* Система программирования "Алгол-68 – Эльбрус" – Препринт № 14/НФ ИТМ и ВТ. – Новосибирск, 1985. – 36 с.
14. *Вельбицкий И.В.* Технология программирования. – Киев:Техніка,1984.
15. *Вирт Н.* Модула–2// Алгоритмы и алгоритмические языки. Языки программирования. – М.: Наука, 1985. – С. 3–46.
16. *Вдовкин С.В.* Обработка абстрактных исключительных ситуаций в системе Клу-Эльбрус // 7-я Всесоюзная школа-семинар "Параллельное программирование и высокопроизводительные системы": Тез. докл. – Киев: ИК АН УССР, 1986. – С. 91–92.
17. *Вдовкин С.В.* Особенности трансляции выражений в МВК "Эльбрус"// II Всесоюзная конференция "Автоматизация произв. ППП и трансляторов": Тез. докл. – Таллин: ИК АН ЭССР, 1983. – С. 126–128.
18. *Вдовкин С.В., Кубенский А.А., Сафонов В.О.* Транслятор Клу-Эльбрус// Программное обеспечение вычислительных комплексов новой архитектуры. – Новосибирск: ВЦ СО АН СССР, 1986. – С. 32–41.

19. *Волконский В.Ю., Пентковский В.М.* Универсальный интерфейс компонентов системы программирования МВК "Эльбрус". – Препринт № 28/ИТМ и ВТ АН СССР. – М., 1980. – 39 с.
20. *Глушков В.М.* и др. Программное обеспечение ЭВМ МИР-1 и МИР-2, т. 1. – Киев: Наукова думка, 1976.
21. *Головкин Б.А.* Многопроцессорные вычислительные комплексы "Эльбрус" (обзор) // Программирование. – 1986. – № 4. – С. 76–87; № 5. – С. 77–83.
22. *Гололобов В.И., Чеблаков Б.Г., Чинин Г.Д.* Описание языка ЯРМО: Машинно-независимое ядро. Макросредства. – Новосибирск: ВЦ СО АН СССР, 1980. – Препринты № 247. – 43 с.; № 248 – 36 с.
23. *Грисуолд Р., Поудж Дж., Полонски И.* Язык программирования СНОБОЛ-4. – М.: Мир, 1980. – 268 с.
24. *Грогоно П.* Программирование на языке Паскаль. – М.: Мир, 1982.
25. *Грэхем Р.* Практический курс языка Паскаль для микроЭВМ. – М.: Радио и связь, 1986. – 200 с.
26. *Гутман А.А., Марков В.А.* Инструментальный комплекс ТЕМП для моделирования МВК "Эльбрус" на БЭСМ-6// Управляющие системы и машины. – 1982. – № 2. – С. 95–97.
27. *Дал У., Мюрхауг Б., Нюгорд К.* Симула-67 – универсальный язык программирования. – М.: Мир, 1969. – 100с.
28. *Дзержинский Ф.Я.* Язык для проектирования структурированных программ // Алгоритмы и организация решения экономических задач, вып. 14. – М.: Статистика. – 1980. – С. 83–96.
29. *Дженибалаев Х.Д.* и др. Транслятор языка Симула-67 в среде МВК "Эльбрус". – Препринт № 17/ НФ ИТМ и ВТ. – Новосибирск, 1985. – 21 с.
30. *Дмитриева М.В., Косовский Н.К., Лаврищева В.А.* Реализация языка Рефал на МВК "Эльбрус" // Программное обеспечение вычислительных комплексов новой архитектуры. – Новосибирск: ВЦ СО АН СССР, 1986. – С. 46–49.
31. *Добров А.Д., Пентковский В.М., Приказчикова М.С., Чижова В.С.* Особенности комплексации программ в системе программирования МВК "Эльбрус". Управляющие системы и машины. – 1982. – № 4. – С. 92–96.
32. Единая система ЭВМ. Система программирования. Язык программирования Паскаль: Руководство программиста. – Ереван: ЕрНУЦ "Алгоритм", 1983. – 56 с.
33. *Замулин А.В.* Язык программирования АТЛАНТ. Предварительное сообщение. – Препринт № 654 / ВЦ СО АН СССР. – Новосибирск, 1986. – 46 с.
34. *Замулин А.В., Скопин И.Н.* Типы данных в языках программирования // Прикладная информатика. Вып. 2 (7). – М.: Финансы и статистика, 1984. – С. 68–92.
35. *Зиглер К.* Методы проектирования программных систем. – М.: Мир, 1985. – 328 с.
36. *Йенсен К., Вирт Н.* Паскаль: руководство для пользователя и описание языка. – М.: Мир, 1982. – 152 с.
37. *Капустина Е.Н., Лавров С.С., Селюн М.И.* Схема расширений и основные принципы реализации аппарата процедур языка Паскаль в языке АБВ// Обработка символьной информации. Вып. 4. – М.: ВЦ АН СССР, 1978. – С. 5–10.
38. *Кириллин В.А.* ФОРТ-система для терминального комплекса ЕС-7970// Аппаратные и программные средства применения ЭВМ в учебном процессе: Материалы конференции "Применение ЭВМ в вузе". – Л.: Изд-во ЛГУ, 1984. – С. 41–42.
39. *Клименко Т.И.* и др. Пакет прикладных программ по численному анализу ФОРПАК для МВК "Эльбрус". – Препринт № 18/НФ ИТМ и ВТ. – Новосибирск, 1985. – 11 с.
40. *Кубенский А.А.* О реализации управления в современных языках программирования// Программирование. – 1984. – № 5. – С. 47–55.
41. *Кубенский А.А., Сафонов В.О.* Возможности языка АБВ и его реализация// Программирование. – 1982. – № 3. – С. 64–72.
42. *Лавров С.С.* Введение в программирование. – М.: Наука, 1977.
43. *Лавров С.С.* Расширяемость языков. Подходы и практика // Прикладная информатика. Вып. 2 (7). – М.: Финансы и статистика, 1984. – С. 17–22.
44. *Лавров С.С.* Языковая основа применений ЭВМ // Журнал вычислительной математики и математической физики. – 1971. – Т. 11, № 2. – С. 498–504.
45. *Лавров С.С., Капустина Е.Н., Селюн М.И.* Расширяемый алгоритмический язык АБВ // Обработка символьной информации. Вып. 3. – М.: ВЦ АН СССР, 1976. – С. 5–53.

46. *Лавров С.С., Силагадзе Г.С.* Автоматическая обработка данных. Язык Лисп и его реализация. – М.: Наука, 1978. – 176 с.
47. *Липаев В.В.* Промышленная технология разработки программных средств для встраиваемых микроЭВМ (технология ПРОМЕТЕЙ)// Вычислительная техника социалистических стран. Вып. 18. – М.: Финансы и статистика, 1985. – С. 54–65.
48. *Майерс Г.* Архитектура современных ЭВМ, т. 1. – М.: Мир. 1985. – 363 с.
49. *Мамай А.К.* Командные языки и методы их переноса на МВК "Эльбрус" // Проектирование и создание многомашинных и многопроцессорных систем для АСУ народного хозяйства: Материалы семинара. – М.: МДНТП, 1984. – С. 139–144.
50. *Окольнишников В.В., Шелехов В.И.* Архитектура системы Алгол-Эльбрус// Многопроцессорные вычислительные системы и их математическое обеспечение. – Новосибирск: ВЦ СО АН СССР. – 1982. – С. 59–66.
51. Операционная система ОС РВ. Система программирования. Язык программирования Паскаль: Руководство программиста. – М.: ИНЭУМ, 1984.
52. *Пентковский В.М.* Автокод Эльбрус. Принципы построения языка и руководство к пользованию / Под ред. Ершова А.П. – М.: Наука, 1982. – 352 с.
53. *Пентковский В.М.* Вопросы управления программными конфигурациями и версиями. – Препринт № 10/ИТМ и ВТ АН СССР. – М., 1985. – 33 с.
54. *Пентковский В.М., Синдеев В.П.* Использование расширяемого языка высокого уровня для определения специализированных языков управления системами// Прикладная информатика. Вып. 2 (7). – М.: Финансы и статистика, 1984. – С. 93–102.
55. Пересмотренное сообщение об Алголе-68: Пер. с англ. Берса А.А. – М.: Мир. 1979. – 533 с.
56. *Пирин С.И.* Язык Паскаль-монитор и его использование. – М.: ВЦ АН СССР, 1978. – 46 с.
57. *Рейтсакас А.А.* Реализация языка ЛИСП для МВК "Эльбрус"// Программирование. – 1984. – № 1. – С. 53–57.
58. *Ренделл Б., Рассел Л.* Реализация Алгола 60. – М: Мир, 1967.
59. *Рябинин В.Н., Самойлов В.Ю.* Диалоговый транслятор с Алгола-60 для МВК "Эльбрус"// Программирование. – 1982. – № 4. – С. 83–88.
60. *Сафонов В.О.* Автокод Эльбрус: Учеб. пособие. – Л.: Изд-во ЛГУ, 1982. – 92 с.
61. *Сафонов В.О.* Вычисление семантических атрибутов системы определений за один просмотр// Всесоюзная конференция по методам трансляции: Тез. докл. – Новосибирск: ВЦ СО АН СССР. – 1981. – С. 106–108.
62. *Сафонов В.О.* Использование архитектуры МВК "Эльбрус" для эффективности реализации динамических языков // 7-я Всесоюзная школа-семинар. "Параллельное программирование и высокопроизв. системы": Тез. докл, Киев: ИК АН УССР. – 1986. – С. 141–142.
63. *Сафонов В.О.* Некоторые методы организации структур данных в трансляторах: Дис. ... канд. физ.-мат. наук. – Л., 1981.
64. *Сафонов В.О.* Транслятор Паскаль-Эльбрус: основные черты и состояние// Аппаратные и программные средства применения ЭВМ в учебном процессе: Материалы конференции "Применение ЭВМ в вузе". – Л.: Изд-во ЛГУ, 1984. – С. 58–59.
65. *Сафонов В.О.* Применение технологических принципов объектно-ориентированного модульного программирования при разработке трансляторов// III Всесоюзная конференция "Автоматизация производства систем программирования": Тез. докл. – Таллин: ИК АН ЭССР, 1986. – С. 47–49.
66. *Сафонов В.О.* Реализация типов в Паскаль-трансляторе// Языки и системы программирования. – Новосибирск: ВЦ СО АН СССР, 1981. – С. 21–31.
67. *Сафонов В.О.* Технологические и практические аспекты разработки системного программного обеспечения МВК "Эльбрус". – Препринт/ НФ ИТМ и ВТ. – Новосибирск, 1988. – 36 с.
68. *Сафонов В.О.* Технологические принципы создания программ обработки сложных структур данных // II Всесоюзная конференция "Технология программирования": Тез. докл. Т. 1. – Киев: ИК АН УССР, 1986. – С. 161–163.
69. *Сафонов В.О., Вдовкин С.В.* Абстрактные типы данных: технология программирования на языке Клу, реализация для МВК "Эльбрус", использование в клу-трансляторе// Программирование. – 1988. – № 6.

70. *Селюн М.И.* О реализации последовательного действия в языке АБВ// Обработка символьной информации. Вып. 5. – М.: ВЦ АН СССР, 1984. – С. 87–94.
71. *Смертин А.Н.* Организация интерпретатора динамического языка символьной обработки Снобол-4: Дис. канд. физ.-мат. наук. – Л., 1985.
72. *Тыугу Э.Х.* Концептуальное программирование. – М.: Наука, 1984. – 255 с.
73. *Фоминых Н.Ф.* Принципы работы и возможности интерпретатора АК ЛГУ // Архитектура и программное оснащение цифровых систем. М.: МГУ, 1984. – С. 32–37.
74. *Форсайт Р.* Паскаль для всех. – М.: Машиностроение, 1986. – 287 с.
75. *Фуксман А.Л.* Технологические аспекты создания программых систем. – М.: Статистика, 1979.
76. *Хоар Ч.* Доказательство корректности представления данных// Данные в языках программирования. – М.: Мир, 1982. – С. 54–67.
77. *Шмундак А.С.* Система МИС: абстрактная машина. – Препринт/ ИК АН ЭССР. – Таллин, 1982. – 33 с.
78. Экспертные системы. Принципы работы и примеры/ Под ред. Форсайта Р. – М.: Радио и связь. 1987. – 225 с.
79. *Ackerman B.W.* et al. VAL – a value-oriented algorithmic language. – MIT/LCS/TR, Cambridge, Mass., 1979.
80. *Addyman T.* et al. Draft report on the programming language PASCAL//Software – Practice and experience. – 1979. – Vol. 9, No 5. P. 385–424.
81. Computer programming language PASCAL. International Standard. – ISO 7185:1983.
82. FORTH-83 Standard. A publication of the FORTH Interest Group. – Mountain Press Rewiew. – August 1983.
83. *Liskov B.* et al. CLU Reference Manual// Lecture Notes in Computer Science. – Springer Verlag, V. 114. – 1981.
84. *Moffat D.V.* A categorized PASCAL bibliography (June 1980)// SIGPLAN Notices. – 1980. – Vol. 14, No 10 – P. 63–75.
85. *Teitelman W.* INTERLISP Reference Manual. Palo Alto: Xerox Palo Alto Research Center, 1974.
86. *Wegbreit B.* The treatment of data types in EL/1 //Communications of the ACM. – Vol. 17, No 5. – P. 251–264.
87. *Wirth N.* Modula: a language for modular multiprogramming// Software – Practice and experience. – Vol. 7, No. 7. – P. 3–35.
88. *Wirth N.* Programming in Modula-2/3rd corrected ed. – Springer-Verlag, 1985. – P. 202.
89. *Wirth N.* The programming language PASCAL//Acta Informatica. – Vol. 10, No. 1. – P. 35–63.

Научное издание

САФОНОВ Владимир Олегович

**ЯЗЫКИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ
В СИСТЕМЕ "ЭЛЬБРУС"**

Заведующий редакцией *О.И. Сухова*
Редакторы *В.М. Ревзина, Е.Ю. Юракова*
Художественный редактор *Т.Н. Кольченко*
Технический редактор *С.В. Геворкян, М.И. Мешкова*
Корректоры *Т.С. Родионова, Н.П. Круглова, Т.В. Обод, Т.А. Печко*

Набор осуществлен в издательстве
на наборно-печатающих автоматах

ИБ № 32847

Сдано в набор 03.01.89. Подписано к печати 19.04.89. Т–08973
Формат 60 X 90/16. Бумага офсетная
Гарнитура Пресс-Роман. Печать офсетная
Усл.печ.л. 24,50. Усл.кр.-отт. 24,50. Уч.-изд.л. 28,19
Тираж 27000 экз. Тип. зак. 583 Цена 2 р. 10 к.

Ордена Трудового Красного Знамени
издательство "Наука"
Главная редакция физико-математической литературы
117071 Москва В-71, Ленинский проспект, 15

Четвертая типография издательства "Наука"
630077 г. Новосибирск-77, ул. Станиславского, 25

ИЗДАТЕЛЬСТВО "НАУКА"

**ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ**

ВЫЙДЕТ В СВЕТ В 1990 Г.:

Дьяконов В.П. **ФОРТ-системы программирования персональных ЭВМ**

В книге детально описываются версии, оперирующие с числами с плавающей запятой. Даются основы программирования на ФОРТе. Приводятся свыше 400 практических примеров расширения версий языка и разработки комплексов прикладных программ, охватывающих реализацию массовых математических, научно-технических, учебных расчетов и различных системных функций. Показываются возможности ПЭВМ: реализация цветной и ЛОГО-графики на ФОРТе, создание звуковых сигналов, изменение алфавита, построение графиков функций и т.д.

Для инженеров и студентов вузов.

Заказы на книгу принимаются всеми магазинами Союзкниги и Академкниги, распространяющими физико-математическую литературу.

В Аннотированном тематическом плане выпуска литературы на 1990 г. книга В.П. Дьяконова занимает позицию № 176.