БИБЛИОТЕЧКА ПРОГРАММИСТА

В. М. ПЕНТКОВСКИЙ

Автокод эльбрус

БИБЛИОТЕЧКА
ПРОГРАММИСТА

В. М. ПЕНТКОВСКИЙ

АВТОКОД ЭЛЬБРУС

ЭЛЬ-76

ПРИНЦИПЫ ПОСТРОЕНИЯ ЯЗЫКА И РУКОВОДСТВО К ПОЛЬЗОВАНИЮ

Под редакцией А. П. ЕРШОВА

МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1982

22.18
П 25
УДК 519.6

Автокод эльбрус. Принципы построения языка и руководство к пользованию. В. М. Пентковский.— М.: Наука. Главная редакция физико-математической литературы, 1982.— 352 с.

Книга знакомит с общими концепциями, положенными в основу многопроцессорной системы Эльбрус, и ролью базового языка автокод в системе. Дается полное описание языка, а также определяются технические детали, необходимые для всех уровней практического программирования на автокоде.

Для программистов и читателей, интересующихся современными тенденциями в развитии вычислительной техники.

Рис. 12 + 2 клише. Библ. 34 назв.

П $\frac{1501000000-130}{053(02)-82}$ КБ 9-29—82

ОГЛАВЛЕНИЕ

ОТ РЕДАКТОРА

С выходом в свет этой книги знание о многопроцессорном вычислительном комплексе (МВК) Эльбрус станет общим достоянием. Эта система завершает серию МЭСМ, БЭСМ, М-20, БЭСМ-6 — разработок, каждая из которых, аккумулируя могучие творческие импульсы выдающегося ученого и конструктора академика Сергея Алексеевича Лебедева и ведя счет поколениям ЭВМ, становилась событием в развитии отечественной вычислительной техники.

Мы позволили себе выражение «завершает», потому что «Эльбрус» — это последняя машина, исходные данные и принципиальные положения которой были разработаны С. А. Лебедевым. Можно сказать, безо всяких преувеличений, что проект МВК Эльбрус стал его научным завещанием. Тем более отрадно видеть, что все наиболее интересные научные идеи, присущие этому проекту, нашли свое достойное воплощение уже в первой модели МВК Эльбрус-1.

В архитектуре «Эльбруса» получила замечательное развитие восходящая еще к первому поколению ЭВМ идея мощной универсальной вычислительной машины общего назначения. Всего две модели в серии благодаря модульности и многопроцессорности охватывают два порядка на шкале производительности ЭВМ. Мощные средства контроля и другие конструктивные решения позволяют обойтись одним стандартом надежности, удовлетворяющим широкому спектру применений. Базовая операционная система без каких бы то ни было опций и надстроек обеспечивает самые разные режимы работы и их сочетание: пакетное мультипрограммирование, разделение времени, многопроцессорная обработка, горячее резервирование и работа в сети. Единый архив обслуживает как задачи пользователей, так и саму операционную систему.

«Эльбрус», однако, интересен не только преемственностью идей. С его появлением в практику отечественного вычислительного дела прочно вошел принцип приоритета методов программирования перед чисто конструкторскими задачами построения машины. Аппаратная поддержка базовых конструкций алгоритмических языков, работа с именами, концепция математической памяти — все это свидетельствует о радикальном продвижении аппаратуры в сторону средств программирования. Не менее важным новшеством стало опережающее развитие программной модели ЭВМ по отношению к ее физическому воплощению. Инструментальные комплексы вдохнули

жизнь и еще не созданное творение, а это помогло не только заблаговременной разработке программного обеспечения, но и снабдило инженеров-проектировщиков неведомой ранее обратной связью.

Все эти и многие другие неназванные, но не менее интересные свойства МВК находят свое воплощение в языке системы — автокоде эльбрус. Именно изобразительные средства автокода раскрывают особенности архитектуры комплекса. Не имеет смысла предвосхищать в кратком вступлении изложение языка, но стоит подчеркнуть одно его принципиальное свойство, которое позволяет удовлетворять критерию универсальности без ощутимой потери эффективности,— это его динамичность. С одной стороны, динамичность поддерживается аппаратными средствами контроля, в частности тегированной памятью, а с другой стороны, динамичность позволяет без избыточности отслеживать ход вычислительного процесса.

В контактах редактора с автором последний как-то обратил внимание на то, что некоторые читатели рукописи критикуют выбор слова «автокод», полагая это слово архаичным и затушевывающим сущность языка. Хотел бы сказать несколько слов в защиту этого термина, который лично мне очень нравится. Наиболее прочно слово «автокод» связано с серией «доалголовских» трансляторов, создававшихся в Англии, в частности А. Брукером. Эти трансляторы назывались «автокодами» и примерно соответствовали нашим «программирующим программам» 50-х годов. Те автокоды были к внутреннему машинному языку ЭВМ 1-го и 2-го поколений примерно в той же пропорции, что ныне автокод эльбрус — к его внутреннему машинному представлению. Сохранение слова «автокод» в применении к «Эльбрусу» — это невинная бравада, подчеркивающая резкое повышение уровня машинных языков.

Не менее важным итогом проекта «Эльбрус», нежели создание машины и его программного обеспечения, явилось создание замечательного коллектива инженеров, математиков и программистов, выполнивших эту выдающуюся разработку. Этот коллектив объединяет уникальный опыт первых учеников академика С. А. Лебедева с энергией и образованностью выпускников МГУ и МФТИ, для которых «Эльбрус» стал их первой серьезной работой.

Эта книга является учебником по языку. Сам автокод еще продолжает развиваться. Тем не менее она дает развернутое представление о принципах построения языка и может служить достоверным руководством к пользованию автокодом эльбрус.

А. П. Ершов

ПРЕДИСЛОВИЕ

В предлагаемой книге изложены принципы построения языка программирования автокод эльбрус *), дано его полное описание, а также приведен ряд дополнительных материалов, необходимых в практической работе с языком. Таким образом, книга, в основном, посвящена вопросам программирования в системе Эльбрус и вопросам развития алгоритмических языков. Можно отметить, что подобная специализация изложения не позволяет исчерпывающим образом охарактеризовать особенности данной системы. Поэтому выходу книги должно было бы предшествовать издание, достаточно полно и всесторонне отражающее тематику «Эльбруса» в целом. Чтобы хоть в какой-то степени заполнить этот пробел, в предисловии и в начале первой главы приведены некоторые сведения наиболее общего характера, позволяющие читателю получить представление о той среде, в которой разрабатывался язык.

Вычислительная система Эльбрус разработана в Институте точной механики и вычислительной техники АН СССР им. С. А. Лебедева (ИТМ и ВТ) коллективом инженеров, конструкторов и программистов под руководством В. С. Бурцева. Разработка системы в целом носила интегральный характер, т. е. одновременно разрабатывались конструктивные основы и архитектура машины, структура операционной системы, система программирования, включающая трансляторы распространенных языков высокого уровня, и язык общения с системой — автокод эльбрус, также являющийся языком высокого уровня. Как видим, под термином «вычислительная система» подразумевается единое по замыслу образование, состоящее из ЭВМ — высокопроизводительного многопроцессорного вычислительного

*) Когда книга уже находилась в наборе, было принято решение о присвоении языку названия «эль-76». Цифры в названии обозначают год разработки эксплуатируемой версии языка.

комплекса (МВК) Эльбрус — и общего системного программного обеспечения (ОСПО) этого комплекса. Разработка ОСПО в части трансляторов и систем трансляторов алгола, фортрана и других основных языков программирования проводилась в Новосибирском филиале (НФ) ИТМ и ВТ.

Положительный опыт предшествующих проектов, в том числе опыт ведущих отечественных разработок в области вычислительных машин, операционных систем и систем программирования, обеспечил для МВК и ОСПО Эльбрус хорошую базу, которую можно было рассматривать как основу для дальнейшего развития. В этой связи надо прежде всего упомянуть серию работ по созданию высокопроизводительных универсальных ЭВМ, приведших к построению ЭВМ БЭСМ-6, а также принцип структурной интерпретации языков высокого уровня, выдвинутый В. М. Глушковым и нашедший свое начальное воплощение в ЭВМ серии МИР и проекте машины Украина [34].

В системе Эльбрус решение проблем программного обеспечения существенно связано с решением задач инженерного характера, и провести границу между ними сложно. Однако, в связи с конкретностью назначения книги, система (и, в частности, аппаратура) рассматривается в ней с точки зрения вопросов программирования. Надо, однако, учитывать, что за этим стоит целый ряд важных инженерных и конструкторских решений, пояснить которые в рамках данной работы, к сожалению, не представляется возможным.

В первой главе книги обсуждается взаимосвязь принципов построения системы Эльбрус и основных аспектов разработки автокода. В начале ее определены цели создания системы, причем основное внимание уделено вопросу развития средств программирования и полному переходу на языки высокого уровня. Далее анализируются те особенности традиционных языков, которые ограничивают область их применения, даются характеристики автокода и связанные с ними черты МВК Эльбрус и операционной системы. В заключение рассматривается вопрос об универсальном использовании данного языка высокого уровня, как единого средства общения программиста с вычислительной системой, применимого одновременно как для системного и проблемного программирования, так и для программирования заданий.

Вторая и третья главы содержат полное описание языка. Во второй главе вводится ядро автокода: описания, операторы и выражения. Третья глава целиком посвящена вопросам взаимодействия программы с внешними объектами (с файлами,

с архивом, с другими программами). Четвертая и пятая главы, а также приложения содержат справочно-информационный материал, который может понадобиться в практической работе с языком (атрибуты объектов, номенклатура ошибок взаимодействия с внешними объектами и средства обработки ошибок, внутренняя кодировка символов, синтаксис языка и пр.).

Описание языка складывалось в течение ряда лет в процессе разработки системы. Оно выполняло роль спецификации языка при разработке трансляторов, служило в качестве проектной спецификации логического интерфейса тех компонентов системы, которые представлены в языке соответствующими конструкциями, а также использовалось как руководство к программированию. Первые два обстоятельства позволяют считать описание достаточно полным и достоверным; они же объясняют форму и подробность изложения. Перед публикацией текст был переработан в стилистическом отношении, а также в части используемой терминологии. При этом исходное назначение описания было сохранено.

Автор не ставил перед собой цель разработать учебник языка. Однако, учитывая, что в настоящее время отсутствует подобное пособие, изданное массовым тиражом *), предпринят ряд мер, которые, возможно, облегчат изучение языка. Для этого сделано следующее:

1) Центральным разделам описания предпослано краткое вступление, в котором на примерах объясняется взаимосвязь конструкций и понятий, вводимых в последующем тексте этих разделов.

2) Основное описание разбито на две части (главы 2 и 3), причем во вторую часть (гл. 3) вынесено описание таких средств языка, которые могут понадобиться, в основном, при программировании алгоритмов, активно взаимодействующих с внешними объектами.

3) Материал каждой главы разбит на основной и второстепенный (т. е. такой, который, возможно, не понадобится на первых стадиях изучения и практического применения языка). Абзацы и параграфы, содержащие второстепенные сведения, набраны более плотно, нежели основной текст; в начале и в конце таких разделов находится символ *. Предполагается, что на первом этапе знакомства с языком будет достаточно основных сведений первой части (§§ 1—13 гл. 2, всего около 80 стр.).

---

*) В ВЦ ЛГУ выпущен препринт В. О. Сафонова, являющийся учебным пособием по автокоду, в котором в доступной и наглядной форме изложено ядро языка.

Сюда, в частности, входят простейшие способы форматного ввода/вывода и элементарные приемы составления пакетных заданий. Примеры, приведенные в конце этой части (§ 20 гл. 2), также базируются на материале основного текста. Иными словами, тем самым как бы дается «подсказка»: разделы и традиционные примеры, входящие в основной текст гл. 2 подобраны так, чтобы программист, знакомый с численным и логическим программированием на алголоподобных языках, мог, не изучая всего описания, провести аналогию между автокодом и знакомым ему языком программирования. В силу универсальности языка этих же сведений достаточно, чтобы написать не только простую программу, но и пакетное задание для ее запуска. При необходимости программа может содержать операторы обмена, а пакетное задание — определение входных и выходных файлов.

Отметим, что несмотря на подобное разграничение последовательность изложения и полнота разделов сохранены (кроме несущественных отклонений от этого принципа в §§ 12, 13 гл. 2). При необходимости описание можно читать подряд, не принимая во внимание разницу в наборе отдельных участков текста.

По-видимому, имеет смысл сказать несколько слов об основных этапах развития языка, поскольку его история отражает эволюцию системы в целом. Предварительный вариант системы команд и языка появился в 1972—1973 гг. После выпуска описания первой версии автокода в 1973 г. были предприняты первые эксперименты по использованию языка для программирования операционной системы и трансляторов некоторых алгоритмических языков. На основании полученного опыта программирования и опыта реализации языка было признано целесообразным внести некоторые изменения в систему команд и в автокод. В частности, стало очевидным, что есть возможность дополнительно повысить уровень языка, не снижая при этом эффективность рабочих программ. В связи с этим в 1976—1977 гг. была введена новая версия системы команд и выпущено описание новой версии автокода. Этот этап развития системы особенно отчетливо свидетельствует о взаимовлиянии вопросов разработки аппаратуры, программного обеспечения и языка.

Начиная с 1977 г. ядро языка в достаточной степени стабилизировалось, но, по мере развития системы, его возможности регулярно расширялись. Наиболее значительным этапом было введение в язык средств взаимодействия с внешними объектами. В книге представлены те средства языка, которые отработаны на настоящий момент. Однако автокод продолжает раз-

виваться. В частности, проходят опытную эксплуатацию средства «определяемого синтаксиса», которые позволяют, при необходимости, вводить для каждого достаточно большого программного комплекса (в частности, для пакетов прикладных программ) собственный входной язык. Предполагаются и другие расширения автокода.

* * *

В предлагаемом языке и книге отражены тем или иным образом результаты труда большого коллектива инженеров и программистов. Поэтому автор считает необходимым назвать, по крайней мере, тех из них, кто оказывал наибольшее влияние на формирование и практическое внедрение языка, а также тех, кто помогал ему при подготовке публикации.

В первую очередь, надо подчеркнуть, что за разработкой языка стоит разработка тех функциональных возможностей машины и операционной системы, которые связаны с развитием и реализацией фундаментальных понятий алгоритмических языков и процессов обработки информации. Идеи, составившие основу этой части проекта «Эльбрус», и, в том числе, положения, обусловливающие основные черты автокода, были предложены руководителем разработки архитектуры машины и программного обеспечения Б. А. Бабаяном. Ведущими разработчиками компонентов ОСПО, определивших возможности языка, являются: А. П. Иванов (управление памятью), С. В. Семенихин (управление процессами и задачами в многопроцессорном мультипрограммном режиме: пакетная обработка и режим разделения времени), В. П. Торчигин (управление файлами); разработка и реализация языка, как и реализация всего ОСПО, были бы невозможны без энергичной поддержки и руководства А. Л. Плоткина. Автор книги выполнял разработку языка и его спецификаций и вел работы по системе программирования в части автокод-транслятора и многоязыковых компонентов (комплексация программ, диагностика и отладка и др.).

Разработка языка происходила в тесном контакте с инженерами — разработчиками МВК Эльбрус, с создателями трансляторов автокода и с пользователями языка. Этими первыми и весьма требовательными пользователями, написавшими и отладившими около 200—300 тысяч строк достаточно сложных программ на автокоде, были разработчики ОСПО, в том числе квалифицированный коллектив системных программистов НФ ИТМ и ВТ, руководимый Г. Д. Чининым. Возможность непосредственного взаимодействия со всеми создателями системы, высказанные ими предложения и критика недостатков крайне

помогали в работе над языком и его описанием. Хочется. в частности, отметить важный вклад А. Н. Белостоцкого, В. Ю. Волконского, Н. Б. Малышева, Ю. С. Румянцева, Б. П. Синдеева и Е. Н. Черниса, а также И. Ф. Лесовой, В. А. Маркова, В. Н. Поливанова и Н. А. Шишовой из НФ ИТМ и ВТ, выявивших в процессе реализации языка много его слабых сторон и внесших ряд ценных предложений. Всем им автор приносит свою искреннюю благодарность, а также тем, кто способствовал появлению этой книги. Прежде всего Н. В. Мартыновой за подготовку примеров § 20 гл. 2 и ряда приложений; Т. П. Новиковой, оказавшей неоценимую помощь при подготовке рукописи, и помогавшим ей С. К. Владимировой, Р. А. Родиной и Е. М. Тюриной.

Автор глубоко признателен редактору книги — одному из основателей и руководителей НФ ИТМ и ВТ А. П. Ершову за плодотворные обсуждения концептуальных аспектов разработки, позволившие автору глубже раскрыть принципы построения языка, за критические замечания по первоначальному варианту рукописи, которые послужили основой для ее переработки, а также за постоянное внимание и активную поддержку работ, связанных с программным обеспечением «Эльбруса»; А. Л. Плоткину за большую помощь на всех этапах подготовки книги; В. Ю. Волконскому, В. И. Головачу (ИПУ), В. Ш. Кауфману (МГУ) и А. М. Степанову, взявшим на себя труд прочесть первоначальный вариант первой главы и высказавшим многочисленные полезные замечания и советы; В. О. Сафонову (ЛГУ) за идею простого и наглядного примера на динамику типов; В. А. Андрющенко (МГУ), В. В. Бролю, С. В. Веретенникову, Ю. С. Румянцеву и В. Б. Яковлеву, чьи ценные замечания по описанию языка автор постарался, по мере своих возможностей, учесть при подготовке рукописи.

Особенно я обязан Б. А. Бабаяну. Именно в ходе многочисленных бесед с Борисом Арташесовичем, имевших место на протяжении всей совместной работы над автокодом, формировалось мое представление о назначении и средствах языка, а также складывался и отрабатывался материал разделов книги, посвященных принципам его построения.

В. Пентковский

ГЛАВА 1

ОСНОВНЫЕ КОНЦЕПЦИИ ЯЗЫКА И ЕГО МЕСТО В СИСТЕМЕ ЭЛЬБРУС

1. Вводные замечания

Язык программирования автокод эльбрус создавался в процессе разработки системы Эльбрус. Основные концепции языка существенно связаны с целями и принципами построения системы в целом. Начнем поэтому с краткого определения состава системы и целей ее разработки.

Цели разработки системы. Система Эльбрус [1] состоит из ЭВМ — многопроцессорного вычислительного комплекса (МВК) Эльбрус — и общего системного программного обеспечения (ОСПО) этой ЭВМ *). Основными компонентами ОСПО являются операционная система и система программирования. Операционная система обеспечивает режим пакетной обработки информации и режим разделения времени. В систему программирования входят трансляторы различных языков высокого уровня (в частности, транслятор автокода) и ряд сопутствующих программ (система динамической диагностики и отладки, комплексатор программ и др.).

При разработке системы Эльбрус преследовались, в основном, три цели:

---

*) Существует семейство машин Эльбрус (МВК Эльбрус-1 и МВК Эльбрус-2), отличающихся по производительности. Машины разрабатывались по общим принципам, имеют единое ОСПО и, в частности, совместимы по автокоду. Поэтому в дальнейшем мы не будем их различать и будем говорить о системе Эльбрус, подразумевая при этом, что в ее состав входит любая из машин семейства.

— повышение производительности системы за счет использования современных методов изготовления ЭВМ и новых архитектурных решений как в части структуры машины, так и в части структуры программного обеспечения;

— развитие средств программирования в направлении повышения удобства использования, эффективности и надежности;

— обеспечение расширяемости системы. Это — требование к системным возможностям: их совокупность должна являться удобной основой для разработки и эксплуатации надстроенных программных комплексов (новых трансляторов, пакетов прикладных программ, баз данных).

Здесь, во введении, основное внимание уделено второй цели — развитию средств программирования. Это сделано по двум причинам. Во-первых, этот вопрос является ключевым в понимании направленности разработки в целом. Во-вторых, его обсуждение позволит определить назначение и принципы разработки автокода.

Автокод эльбрус. Возможности автокода эльбрус будут рассмотрены в §§ 4—6 этой главы. Однако целесообразно предварительно охарактеризовать одно обобщенное свойство автокода, которое следует отнести к числу основных результатов работы над языком.

Автокод можно назвать универсальным языком высокого уровня. Учитывая, что термин «универсальный» используется достаточно широко и имеет различные толкования, надо пояснить, что здесь имеется в виду.

В рамках традиционных систем программист использует минимум два языка: язык программирования, с помощью которого описывается проблемный алгоритм, и язык управления операционной системой (язык управления заданиями), на котором формулируется задание для вызова программы и передачи ей необходимых файлов. В сложных случаях используется третий язык: язык ассемблера. Последний является средством наиболее эффективного и непосредственного взаимодействия как с машиной, так и с операционной системой. Каждый из названных языков выполняет свою роль и ни один не может действенно заменить два других. Например, (а) язык высокого уровня обеспечивает удобство программирования (лаконичность, выразительность, надежность), но, если необходим доступ к конкретным аппаратным ресурсам, он не заменит язык ассемблера, (б) поскольку программы и архивы не являются объектами обработки в языке высокого уровня, он не может играть роль языка управления заданиями, (в) хотя язык

ассемблера формально может быть использован, как универсальный аппарат программирования, фактически подобное применение его затруднено целым рядом обстоятельств; в частности, чтобы использовать ассемблер для управления операционной системой, необходимо знать соглашения о связях между модулями системы, требуемые ресурсы, схему обработки прерываний и другую специфическую информацию. Таким образом, для общения с системой используется до трех языков, причем существенно то, что они, как правило, сильно отличаются как по синтаксическим формам, так и по составу семантического базиса.

В настоящее время существует достаточно распространенная тенденция к повышению уровня языков, предназначенных для машинно-ориентированного и системно-ориентированного программирования. Откладывая подробное обсуждение до § 3, отметим здесь два основных момента:

1) Определенное несоответствие базовых понятий языков высокого уровня, с одной стороны, и архитектуры машины и операционной системы, с другой стороны, затрудняет процесс сближения языков. Здесь требуются не только исследования в области языков, но и развитие структуры всей системы в целом;

2) Ориентация на существующие системы усложняет разработку единого языка, обладающего и свойством универсальности и концептуальной целостностью. Например, используемые в системном программировании машинно-ориентированные языки повышенного уровня содержат в той или иной форме выход на ассемблерный уровень (что принципиально используется при программировании операционных систем) и, кроме того, не применяются широко для программирования заданий. Другой пример. Средства программирования заданий, обладающие характеристиками языков высокого уровня [24], применяются для управления операционной системой, но не используются для описания проблемных алгоритмов.

Возвращаясь к автокоду, отметим особенность данного языка высокого уровня, состоящую в том, что он используется одновременно во всех перечисленных областях программирования. Для определенности скажем, что в системе Эльбрус нет специализованного языка системного программирования. Все названные выше компоненты ОСПО реализованы на автокоде. Кроме того, в системе нет особого языка управления заданиями. Программа задания — это обычная автокод-программа. Наконец, автокод используется и для проблемного программирования, как один из доступных в системе языков высокого

уровня. Поэтому автокод можно рассматривать как единое средство общения программиста с вычислительной системой, т. е. как универсальный язык высокого уровня.

Свойства языка и системы таковы, что, независимо от конкретной прикладной области, описание алгоритма (задания, системного, проблемного) не требует использования средств, выходящих за рамки понятий языков высокого уровня *). Объясняется это тем, что характеристики языка получены не за счет объединения разнородных средств, а путем расширения сферы применимости фундаментальных понятий языков высокого уровня.

Перейдем теперь к вводному обзору следующих вопросов: (а) применение существующих языков высокого уровня, (б) предпосылки, позволяющие расширить возможности языков, и (в) связь разработки системы Эльбрус с проблемами развития средств программирования.

Применение языков программирования. Изменение технологической базы и архитектуры вычислительных комплексов, увеличение их мощности на несколько порядков определило появление принципиально новых возможностей использования ЭВМ. Соответственно возросла сложность задач, решаемых с помощью ЭВМ, и сложность программ, описывающих алгоритмы решения. Наряду с этим возросли требования к удобству общения человека с машиной. Например, в настоящее время трудно представить современную вычислительную систему без многообразия пакетов прикладных программ, без средств диалогового режима работы.

Программы, обеспечивающие эти удобства, достаточно сложны, а их написание и отладка — трудоемкий процесс. В целом следует отметить, что с ростом возможностей ЭВМ растут и трудозатраты, связанные с программированием. В особенности это касается области системных программ, куда мы включаем операционные системы, различные диалоговые пастройки над ними, компоненты систем программирования, пакеты прикладных программ.

Значительное облегчение работы программиста достигается путем использования языков высокого уровня (ЯВУ). Эти язы-

*) В частности, операционная система МВК Эльбрус, обеспечивающая достаточно богатые функциональные возможности (мультипрограммная обработка в мультипроцессорном режиме, файлы, универсальный многоуровневый архив, разделение времени), полностью реализована на языке высокого уровня и не содержит операторов, написанных в машинных кодах.

ки позволяют описывать алгоритмы более адекватно, нежели ассемблерные средства программирования. В результате упрощаются процесс написания программ, их отладка, модификация и развитие.

Отмечая широкое распространение ЯВУ, надо признать и другой факт: полный перевод программирования на ЯВУ еще не произошел. Существует ряд областей, в которых традиционно используются ассемблерные средства программирования (АСП) *). Две основные причины этого явления состоят в следующем:

1. Каждый из распространенных ЯВУ содержит ряд принципиальных ограничений. Например, имя переменной надо с необходимостью статически связывать с типом ее возможных значений. Другой пример: ЯВУ не содержат средств, позволяющих исчерпывающим образом описать обработку таких объектов, которые можно было бы назвать внешними. К ним относятся файлы, программы и другие объекты, расположенные на внешних носителях.

Подобные ограничения обусловлены достаточно серьезными обстоятельствами. Авторы языков стремились, с одной стороны, обеспечить эффективность и надежность программ и, с другой стороны, ориентировались на реализацию в рамках ограниченных возможностей традиционных машин и операционных систем. В условиях этих противоречивых предпосылок компромиссное решение, естественно, состоит в том, чтобы ввести в язык некоторые ограничения.

Чтобы преодолеть эти ограничения, используются АСП, семантика которых определяется конкретной машиной и операционной системой.

2. Несмотря на стремление к эффективности, в ряде случаев не удается получить достаточно удовлетворительное каче-

---

*) Забегая вперед, сделаем следующее пояснение. В последнее время нашли применение (особенно в системном программировании) такие языки, которые можно было бы поместить между ассемблерными языками и ЯВУ. Их изобразительные средства похожи на принятые в ЯВУ, но есть и различного рода ассемблерные характеристики: контроль типов либо не производится, либо есть возможность его отключать, разрешен выход в ассемблерный язык и пр. (подробно см. § 3). Поскольку эти характеристики, в частности, отсутствие контроля, снижают надежность программ, мы будем условно причислять такие языки (точнее — их машинно-ориентированные средства) к АСП.

ство реализации тех или иных конструкций ЯВУ. Кроме того, традиционные машины имеют некоторые специфические возможности, которые не имеют эквивалентов в ЯВУ, но повышают скорость вычислений (например, средства работы с регистрами). Каждое из этих обстоятельств приводит к тому, что для повышения эффективности программы необходимо использовать АСП.

Известно, что один из основных недостатков применения АСП — это снижение надежности программ. Однако, ввиду отсутствия других средств, с этим фактом приходится мириться. Противоречивость ситуации особенно заметна в системном программировании. С одной стороны, системное программирование — это основная область, в которой используются АСП, с другой стороны, это одна из тех областей, где требования к надежности наиболее высокие.

Новые возможности. В настоящее время есть достаточно серьезные основания для того, чтобы сделать следующий шаг в развитии средств программирования и осуществить перевод всех без исключения областей программирования на языки высокого уровня.

В первую очередь отметим предпосылки, связанные с развитием электроники. Снижение стоимости производства электронных компонентов ЭВМ и современные методы микроминиатюризации позволяют, не увеличивая общей стоимости и объема машины, усложнить алгоритмы машинных операций и обогатить их функциональное наполнение. Это, в частности, дает возможность осуществить аппаратную реализацию наиболее общих семантических механизмов ЯВУ.

Отсюда вытекают два очевидных следствия. Во-первых, эффективность использования ЯВУ может быть поднята до такого показателя, при котором отпадает необходимость применения АСП, как средства повышения производительности программы. Во-вторых, можно пересмотреть упоминавшиеся выше ограничения ЯВУ. При соответствующем выборе аппаратно реализованных понятий эти ограничения могут быть в значительной степени устранены. В результате появляется возможность разрабатывать и практически внедрять новые ЯВУ с расширенными возможностями.

Не менее важные предпосылки связаны с развитием операционных систем (ОС). Ранние ОС предназначались для управления всем комплексом устройств, составляющих ЭВМ, а также для повышения эффективности использования машины. Сейчас, наряду с этим, намечается тенденция к повышению уровня семантических понятий, реализованных в ОС. Например,

в развитых ОС вместо средств явного распределения вторичной памяти предоставляется возможность заказывать файл, т. е. объект, обработка которого не зависит от его конкретного местоположения; все вопросы, связанные с распределением физической памяти для файла, решаются системой неявно, без специальных указаний со стороны программы.

Таким образом, тенденции в развитии ОС схожи с тенденциями в развитии аппаратуры: сближение с языками высокого уровня. Как и в случае аппаратуры, такое сближение не только способствует эффективной реализации существующих ЯВУ, но и стимулирует разработку ЯВУ, обеспечивающих новые возможности. Пусть, например, системный способ именования файлов в архиве построен так же, как принятый в ЯВУ способ именования программных объектов. Тогда он достаточно органично может быть встроен в язык. Пользуясь таким ЯВУ, можно описывать различные работы с архивом, не прибегая при этом к АСП или специализированному языку управления заданиями.

Резюмируя, еще раз сформулируем соображения, касающиеся полного перевода программирования на ЯВУ. Современное состояние электронной техники и объем накопленных знаний о разработке, реализации и эксплуатации ЯВУ и ОС позволяют уже сейчас работать в направлении повышения уровня семантических понятий, реализованных системными средствами, т. е. в аппаратуре или в ОС. Это приводит к следующим результатам: а) повышается эффективность реализации существующих ЯВУ и б) становится возможным внедрение ЯВУ, которые по своим возможностям покрывают АСП и предоставляют при этом средства более адекватного программирования.

Система Эльбрус. Сказанное во многом определяет исходные позиции разработки системы Эльбрус. Система в целом характеризуется четко выраженной ориентацией на языки высокого уровня. Систему команд МВК Эльбрус в совокупности с примитивами ОС Эльбрус можно рассматривать как входной язык семантического процессора развитого ЯВУ. Этот язык не совпадает в точности с каким-либо существующим ЯВУ. Скорее можно сказать, что предпринята попытка аппаратно-программной реализации наиболее фундаментальных понятий процессов обработки информации.

В соответствии с исходными посылками некоторые понятия получили дальнейшее обобщение и развитие. Например, механизм защиты, принятый в ЯВУ, основан на так называемом «принципе статической известности идентификаторов» (см. § 2). В системе Эльбрус этот принцип распространен на пространст-

во внешних объектов, т. е. объектов, находящихся в архиве. Такое обобщение имеет целый ряд важных последствий. Одно из них — это такое решение проблемы межпрограммной защиты внешних объектов, которое органически укладывается в ЯВУ. Более подробно это и другие обобщения рассмотрены в § 4.

Место языка в системе. Рассмотрим теперь соотношение автокода и системы Эльбрус. Выше уже упоминалось, что систему можно трактовать как семантический процессор ЯВУ. Поэтому разработку ее примитивов целесообразно проводить совместно с разработкой соответствующего ей ЯВУ. Причем речь идет не о гипотетическом, а реальном, законченном языке. Многочисленные обратные связи, возникающие в процессе совместной разработки, позволяют контролировать приемлемость решений, концептуальную целостность и полноту набора примитивов. Этим определяется теоретическая роль подобного языка. В системе Эльбрус такую роль сыграл язык автокод [1, 2].

Практическая роль автокода вытекает из его характеристик. Оставляя подробное изложение до §§ 4—6, подчеркнем лишь следующую особенность. Автокод является примером ЯВУ с расширенными возможностями. Совокупность свойств языка позволяет использовать его для прикладного и системного программирования и для описания программ управления заданиями. Важно отметить, что широкий диапазон применения получен не за счет механического совмещения разнородных средств, а путем развития и обобщения таких понятий ЯВУ, как «именование», «объект», «процедура». В целом это позволило полностью отказаться от любых разновидностей программирования в кодах и перейти на практике к применению исключительно языка высокого уровня.

Прежде чем перейти к детальному изложению, сделаем еще одно замечание. Поскольку в автокоде достаточно полно и точно отображены возможности системы, обсуждение принципов разработки языка и его характеристик является одновременно вводным обзором свойств системы Эльбрус. Обзор не является всеобъемлющим. Он затрагивает лишь верхний слой из совокупности проблем, рассматривавшихся при разработке системы. Здесь рассматриваются такие вопросы, как статика и динамика типов в ЯВУ, защита и контроль, статическое и динамическое распределение ресурсов, трактовка внешних объектов в ЯВУ, средства обработки ошибок. Вопросы, связанные с распараллеливанием вычислений и телеобработкой, а также подробные технические характеристики МВК Эльбрус и ОСПО не обсуждаются.

2. Основные понятия языков программирования

В этом разделе вводится, по возможности неформально, ряд терминов, используемых в последующем изложении: «объект», «тип», «имя», «обозначение», «описание», «контекст» и пр. Эти понятия составляют, как правило, основу для конструирования языков высокого уровня. Возможности конкретного языка определяются тем, как в нем комбинируются эти понятия, какие между ними связи, какова номенклатура возможных типов.

Объект. В процессе выполнения программа производит действия над объектом. Объекты являются операндами операций; в результате выполнения операции также получается объект. Типичные примеры объектов — числа и логические (или булевские) значения.

Тип. Для каждого конкретного объекта существует ряд допустимых операций; другие операции не определены для этого объекта. Множество объектов, для которых определен один и тот же набор операций, называется множеством объектов определенного типа *). Например, вещественные числа являются объектами типа «вещественное». Для них определены арифметические операции, но не определены, к примеру, логические.

Существует ряд так называемых универсальных операций. Эти операции применимы не к одному, а к нескольким возможным типам операндов. Типичный пример — арифметические операции. Они определены как для целых, так и для вещественных операндов. Это свойство операций еще называют полиморфизмом.

Статика и динамика типов. Понятие «тип» является, безусловно, одним из фундаментальных понятий ЯВУ. Однако языки существенно различаются по средствам управления типами. В основном, для контроля типов используются две стратегии: статическая и динамическая. Языки первой разновидности будем для краткости называть языками класса С, а языки второй разновидности — языками класса Д.

К числу достоинств языков класса Д относят прежде всего простоту программирования и гибкость [4, 11]. Наряду с этим отмечается, что статический контроль типов, хотя и ограничивает возможности языка, существенно увеличивает скорость выполнения программы.

---

*) Такое определение не является исчерпывающим. Однако его, по-видимому, достаточно для первоначального, интуитивного представления о том, что такое тип.

Существует еще третий класс языков — так называемые бестиповые языки *). Особенность их состоит в том, что с объектом не связывается какой-либо определенный тип. Информационное содержимое объекта трактуется в зависимости от того, какая над ним выполняется операция. Например, одна и та же двоичная информация может в арифметических операциях трактоваться как число, а в логических — как булевское значение. По своим синтаксическим формам бестиповый язык может быть во многом схож с ЯВУ. В нем могут быть описания, операторы, выражения, структуры управления (условные предложения, циклы и пр.). Но отсутствие контроля за правильностью трактовки информации сближает эти языки с АСП. В дальнейшем мы, в основном, будем рассматривать языки классов С и Д.

Сделаем замечание относительно хода дальнейшего изложения. Вводимая ниже терминология объясняется в том смысле, который ей придается в языках класса С. Смысл этот иногда несколько отличается от того, который приписывается ей языками класса Д. Различия будут рассмотрены в дальнейшем, в § 4.

И м я. Объекты типа «имя», или имена, играют особую роль. Имя обладает свойством именовать (ссылаться на) другой объект. Пара объектов (имя, именуемый объект) образует переменную. Второй объект из этой пары называется значением переменной. Основные операции над именем — разыменование и присваивание. Результатом разыменования имени переменной является значение этой переменной. Смысл присваивания состоит в том, что оно заменяет текущее значение переменной: в результате его выполнения имя начинает именовать другой объект. При создании имени отводится одна или несколько ячеек памяти, куда будут помещаться именуемые объекты. Собственно имя можно интерпретировать как адрес этих ячеек.

О б о з н а ч е н и е и о п и с а н и е. Чтобы объекты можно было упоминать в тексте программы, вводится ряд средств для «обозначения» объектов. Такие объекты, как числа и логические значения, имеют общепринятые стандартные обозначения. Например, последовательность литер 3.14 обозначает вещественное число 3,14.

С объектом можно связать произвольное обозначение. Для этого используется «описание», в котором указывается обозна-

*) В зависимости от уровня изобразительных средств различают две разновидности бестиповых языков [11]: ассемблерные языки и языки, подобные ЯВУ.

чение и обозначаемый объект. Обычно в качестве обозначений принято использовать идентификаторы. Внешняя форма описаний зависит от особенностей конкретного языка, в частности от способа трактовки типов.

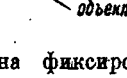
Одно из возможных описаний — описание переменной. Оно связывает обозначение с именем, создаваемым при выполнении описания. В языке алгол-60 это описание выглядит так *):

вещ x, y

Такое же описание в языке паскаль [12] имеет форму:

перем x, y : *вещ*

Теперь отметим наиболее существенную особенность языков класса С: в описании переменной указывается тип ее возможных значений. Формально это трактуется следующим образом. Имя может именовать не любой объект, а лишь объекты заранее предписанного типа. Поэтому считается, что нет типа «имя любого объекта», а есть тип «имя объекта типа Т», где Т — некоторый конкретный тип. Этот тип Т и указывается в описании переменной. Это по существу означает, что тип объекта связывается не с ним самим, а с именем, которое на него ссылается. Связь между именем, именуемым объектом и его типом можно схематически изобразить так:



Двойной линией обозначена фиксированная связь, появляющаяся в момент создания имени. Одинарной линией — переменная связь между именем и объектом. Эта связь является переменной в том смысле, что на конце ее могут появляться различные объекты.

Простые и составные объекты. Такие объекты, как целые и вещественные числа, имена, называют простыми. Тем самым подчеркивается их неделимость. Будем считать, что все простые объекты занимают одну ячейку памяти.

Составные объекты, в отличие от простых, содержат некоторое число компонентов. Простейшим примером является массив. Массив состоит из совокупности линейно упорядоченных имен одинакового типа. Эти имена называются «именами элементов массива». С массивом неразрывно связан простой объект типа «указатель массива». Создание массива заключается

*) Здесь и далее будут использоваться русские эквиваленты служебных слов и стандартных обозначений.

в том, что создаются составляющие его имена, т. е. отводится память. Одновременно создается указатель. Можно считать, что указатель ссылается на массив, или описывает массив. Массив создается в результате выполнения описания. Например, при выполнении описания на языке паскаль:

перем *а* : **массив** [1 .. *n*] **из** *вещ*

происходит следующее. Создается массив и объект типа «указатель массива имен объектов типа вещественное». С этим указателем связывается обозначение *а*. Подобное описание иногда называют описанием массива.

Иной смысл вкладывается в следующее описание *):

перем *р* : ↑ **массив** [1 .. *n*] **из** *вещ*

Здесь создается объект типа «имя объекта типа указатель массива имен объектов типа вещественное», о чем говорит символ ↑; с именем связывается обозначение *р*. Иными словами, это описание переменной-указателя. Ее значениями в разное время могут быть указатели на различные массивы. Например, в результате выполнения оператора:

нов (*р*)

создается массив такого типа, как задано в описании *р*; указатель этого массива становится значением переменной *р*. Здесь мы использовали так называемую процедуру-генератор *нов* (*new*), встроенную в язык.

Как видим, во всех случаях тип объектов, на которые может ссылаться имя (или указатель), фиксируется в момент описания.

Множество типов, вообще говоря, бесконечно. Это видно хотя бы из того, что элемент массива может в свою очередь иметь тип указателя массива. На других примерах мы останавливаться не будем, поскольку для дальнейшего достаточно иметь один факт, приводящий к бесконечному числу типов.

Процедуры и операции. Одной из разновидностей объектов являются процедуры. Для объектов типа «процедура», как и для чисел, вводится некоторый способ обозначения — текст процедуры. Как правило, это изложение алгоритма, за-

*)Здесь и далее используется лексика языка паскаль. Некоторыми особенностями языка мы при этом пренебрегаем (например, после символа ↑ нельзя давать развернутую спецификацию типа, а можно только указывать идентификатор типа). Это сделано для сокращения текстов примеров.

ключенное в определенные ограничители. Возможна, например, такая форма:

(x : $t1$) : $t2$; **начало** ... **конец**

где x — идентификатор формального параметра типа $t1$, $t2$ — тип результата; между служебными словами **начало** и **конец** помещается изложение алгоритма.

Используя описание, процедуру можно связать с произвольно выбранным обозначением:

функция p (x : $t1$) : $t2$; **начало** ... **конец**

Операции можно трактовать как процедуры со специфической формой обозначения (с помощью таких литер, как +, —, * и т. д.) и особой формой вызова. Например, описание операции сложения вещественных чисел может выглядеть так:

функция "+" (u , v : *вещ*) : *вещ* ; **начало** ... **конец**

а вызов —

$x + y$

Контекст. Понятие контекста непосредственно связано с понятием области действия идентификаторов. Чтобы не перегружать изложение, не будем приводить известное правило, определяющее область действия. Можно воспользоваться правилом любого ЯВУ, основанного на принципе статической известности идентификаторов. Последний подразумевает, что область действия определяется статически, т. е. по тексту программы. К классу таких языков относятся и алгол, и симула-67 [8], и modula [18], и ada [21] (в последних трех возможности алгольной блочной структуры расширены, но принцип статической известности выполняется).

Совокупность идентификаторов, описанных вне процедуры и действующих внутри нее, называется контекстом идентификаторов для данной процедуры. Если к контексту идентификаторов присоединить совокупность действующих обозначений для операций, то получим контекст обозначений. Контекст обозначений и совокупность объектов, прямо или косвенно связанных с этими обозначениями, называется просто контекстом процедуры. Говоря «косвенно связаны», мы подразумеваем, что к этой совокупности относятся и те объекты, которые достижимы через посредство промежуточных объектов типа имя или указатель. Таким образом, процедура в процессе выполнения может обрабатывать и использовать только те объекты, которые находятся в ее контексте. Будем говорить, что эти объекты доступны процедуре.

Контекстная защита. Перейдем к одному из наиболее существенных свойств контекста. Поскольку контекст — это совокупность доступных объектов, такое понятие можно считать основополагающим при разработке средств защиты. Напомним, что внутри процедуры неизвестны (не действуют) обозначения объектов, находящихся вне ее контекста. Поэтому она не может упомянуть (назвать) их, а следовательно, не может воспользоваться ими, т. е. они ей недоступны *).

Фразовая структура алгоритма. Предположим, что в процессе разработки процедуры или программы мы руководствуемся методикой пошаговой конкретизации, широко известной по работам из области структурированного программирования. Вкратце она заключается в следующем. Вначале формулируется общее назначение процедуры. Алгоритм процедуры изображается в виде единственного оператора, представляющего собой запись этой формулировки. Пусть он имеет вид P_0. На следующем шаге рассматривается ряд более мелких действий, на которые распадается P_0:

$$P_{11}; P_{12}; \ldots P_{1n}.$$

Это называется конкретизацией абстрактного оператора P_0. Затем конкретизируется каждый из P_{1i} и т. д. В результате многократного повторения процесса конкретизации возникает окончательный текст описания алгоритма на языке программирования. Участки текста, отображающие абстрактные операторы, называются фразами; каждую фразу можно условно обозначить названием соответствующего абстрактного оператора. Вследствие пошаговой конкретизации фразы оказываются вложенными одна в другую. Поэтому будем говорить о рекурсивной фразовой структуре текста программы.

Рекурсивная структура выражена не только в тексте. Она отражается и на процессе выполнения алгоритма. Выполнение фразы P_0 состоит в выполнении фраз P_{1i}; то же самое можно сказать о выполнении каждой из P_{1i} и т. д.

Как известно, отладка и изучение программы существенно упрощается, когда структура вычислений, возникающих в процессе выполнения программы, соответствует фразовой структуре ее текста. Это соответствие обеспечивается если: а) программа не содержит переходов извне фразы вовнутрь, и б) оператор перехода не используется как средство порождения итеративных процессов и условного выполнения операторов.

*) Объекты, переданные в качестве фактических параметров, находятся в контексте процедуры, поскольку они косвенно доступны с помощью обозначений формальных параметров.

3. Некоторые особенности языков высокого уровня

В этом разделе будут рассмотрены отдельные черты распространенных ЯВУ, затрудняющие их использование в ряде приложений. В каждом конкретном случае мы приведем соответствующие примеры и назовем возможные причины, по которым те или иные ограничения вводились в ЯВУ.

Для анализа применимости языка программирования важны такие понятия, как адекватность отображения исходного алгоритма в программу (в ее операторы и структуры данных), надежность, простота программирования. Не определяя эти понятия формально, мы тем не менее попытаемся дать о них интуитивное представление. С этой целью один из вопросов, а именно, статика типов, рассмотрим более подробно, чем остальные. Выбор именно этой тематики обусловлен соображениями наглядности: здесь можно привести обозримые примеры, пользуясь при этом привычной нотацией распространенных ЯВУ. Однако следует учитывать, что названные понятия общезначимы. Они так или иначе связаны и с другими вопросами конструирования ЯВУ.

3.1. Статика типов. Языки класса С постулируют фиксированную связь имени со статически определяемым типом именуемого объекта. Эта связь возникает в момент создания имени; тип именуемого объекта задается в конструкциях, предназначенных для создания имен (в простейшем случае это описания).

Подобная связь является примером так называемых инвариантных соотношений, т. е. таких соотношений между объектами, которые сохраняются в процессе выполнения программы. Особенность, однако, заключается в том, что данное соотношение предписано языком априорно. Независимо от природы алгоритма оно с необходимостью должно удовлетворяться. Такое требование, естественно, ограничивает применимость языка.

Существуют алгоритмы, в которых соотношения между именами и типами именуемых объектов иные. Например, возможны такие связи:

— тип именуемого объекта принадлежит некоторому статически детерминированному множеству типов;

— тип именуемого объекта вычисляется при создании имени и фиксируется;

— тип именуемого объекта может быть любым.

Очевидно, что алгоритмы, обладающие подобными характеристиками, нельзя достаточно адекватно описать, оставаясь в

рамках статической концепции типов. Использование в этих случаях языка класса С может привести к усложнению программы и/или уменьшению надежности.

Пример 1. *Суммирование чисел.* В качестве первого примера, вводящего в проблематику статики типов, рассмотрим простую программу на алголе-60. Эта программа выполняет суммирование чисел.

Пример 1.а.

```
начало
   вещ массив a [0 : 99];
   вещ процедура сумма (b , n);
      значение n ; цел n ; вещ массив b ;
      начало
         вещ c ; цел k;
         c := 0;
         для k := 0 шаг 1 до n цикл
            c := c + b [k] ;
         сумма := c
      конец ;
   ввод (0 , a) ;
   вывод (1 , сумма (a , 99))
конец
```

Программа вводит массив вещественных чисел и выводит их сумму. Предположим, что оператор *ввод* допускает, чтобы среди чисел, вводимых в вещественный массив, содержались не только вещественные числа, но и целые. Тогда последние при вводе будут преобразовываться в вещественные. Причина этого действия чисто формальная. Оно объясняется особенностями языка, а не алгоритма. Пытаясь описать алгоритм средствами алгола-60, мы воспользовались массивом, элементы которого должны (по определению языка) иметь одинаковый тип. Отсюда и возникло преобразование.

Далее, поскольку функция *сумма* специфицирована типом вещественное, результат выводится в виде вещественного числа. Если все вводимые числа — целые, эффект будет тот же самый; числа преобразуются в вещественные и выводится вещественный результат.

Как написать универсальную программу, обеспечивающую вывод целочисленного результата в случае, если все вводимые числа — целые? Эта, на первый взгляд, простая постановка задачи недостаточно хорошо согласуется с принципом статики типов. Действительно, на языках класса С удобно описывать алгоритмы обработки данных, тип которых статически пред-

определен. Однако, если существует некоторая «неопределенность» в типах (пусть даже — минимальная, как в нашем случае), то приходится прибегать к громоздким методам программирования. Воспользуемся, например, техникой объединенных типов алгола-68 *). Напомним, что объединенный тип — это статически детерминированная совокупность некоторого числа типов. Переменная объединенного типа может принимать значение любого типа из этой совокупности. В приведенном примере переменные типа цв могут принимать значения типа целое или вещественное.

Пример 1. б.

```
начало
   вид цв = об (цел , вещ) ; со тип «целое либо
                                вещественное» со
   [0 : 99] цв а ; со массив, каждый элемент которого может
                    принимать либо целое либо вещественное
                    значение со
   проц сумма = (имя [ ] цв в) цв ;
      начало
         цв с := 0 ;
         для к от 0 до вегр в
         цк
            с :=
            выб с в
            (цел ск) : выб в [к] в
               (цел вк) : с + вк, со цел + цел со
               (вещ вк) : с + вк со цел + вещ со
                     быв,
            (вещ ск) : выб в [к] в
               (цел вк) : с + вк , со вещ + цел со
               (вещ вк) : с + вк со вещ + вещ со
                     быв
            быв
         кц ;
         с
      конец ;
      чит (а) ;
      печ (сумма (а))
конец
```

*) Здесь мы воспользуемся языком алгол-68; попытка запрограммировать эту задачу на языке паскаль приводит к еще более громоздкому результату.

Правая часть оператора присваивания внутри цикла — это предложение, выбирающее одну из альтернатив в зависимости от типа выражения, находящегося в заголовке после слова выб. Перед альтернативой находится описание, действующее внутри нее, например (цел *с*): вводит константу *с*, значением которой является значение выражения из заголовка.

Относительно процедуры ввода сделано следующее предположение: если во входном потоке встречается число, имеющее форму целого, то соответствующему элементу массива присваивается значение типа целое, а иначе — типа вещественное *). Похожим образом функционирует процедура вывода.

Обратим внимание на правую часть оператора присваивания, находящегося в теле цикла. Здесь пришлось запрограммировать перебор всех возможных сочетаний операндов операции сложения. Дело в том, что операции языков класса С можно было бы назвать квазиуниверсальными. С одной стороны, они определены для нескольких типов операндов и для всех возможных типов обозначаются одинаково. С другой стороны, при каждом использовании операции в тексте программы следует точно указывать конкретный тип операндов. В соответствии с этим языки устроены так, что при наличии неопределенности типов надо программировать перебор возможных вариантов. Иными словами, динамический анализ и интерпретация типов полностью переносится на программиста. Отсюда и увеличение текста программы. Ниже, при обсуждении причин статики типов, мы рассмотрим, чем обусловлен такой подход.

Пример 2. *Программирование распределения памяти.* Обратимся к проблемам, связанным с обработкой объектов, тип которых статически неизвестен. В качестве примера рассмотрим некоторые детали реализации процедуры создания объектов (типа процедуры *нов* или *new* в языках паскаль, симула-67). Для простоты будем предполагать, что речь идет только о создании массива элементов простого типа (числа, логические, указатели и пр.).

Пусть такая процедура-генератор называется *вснмассив*. Будем считать, что она описана в стандартном окружении про-

*) Эти предположения сделаны для упрощения программы. На самом деле процедуры обмена алгола-68 не допускают объединенных типов, что также является следствием статической концепции. Учет этого фактора привел бы к дополнительному увеличению текста программы.

грамм. Следуя методу описания алгола-68, введем следующее соглашение. Объекты, описанные в стандартном окружении, разбиваются на два класса. Объекты первого класса известны в контексте процедур стандартного окружения, но недоступны для программ. В описании таких объектов будет использоваться знак ?, например:

перем ? *память* **: ...;**

Объекты второго класса доступны везде, и их описание выглядит обычным образом.

Существо рассматриваемых проблем состоит в следующем. Пусть *р* — процедура программы, вызывающая *генмассив*, т. е. осуществляющая заказ массива. При описании *р*, естественно, известен тип элементов заказываемого массива (обозначим его *t1*). В процедуре *генмассив* этот тип статически неизвестен, поскольку *генмассив* — универсальная процедура: она предназначена для создания массива элементов любого простого типа, а таких типов — бесконечное множество (см., например, замечание в § 2 о бесконечном числе типов указателей).

Процедура *генмассив*, выполняя поступивший заказ, выделяет память в некотором «базовом массиве»:

тип ? *t2*=...;
перем ? *память* **: массив** [...] **из** *t2*

Элементы массива имеют тип *t2*. Последний зависит от особенностей работы процедуры *генмассив*, и, вообще говоря, желательно, чтобы он не использовался в программах пользователя.

Указатель на выделенную часть базового массива надо передать из *генмассив* в вызывающую процедуру. Значит где-то (либо в *р*, либо в *генмассив*) должна быть запрограммирована связь *t1* с *t2*. Но, как мы видели, ни в той, ни в другой процедуре эта пара одновременно статически неизвестна. Это один из тех случаев, когда на практике не удается следовать принципу статической известности типов.

Теперь остановимся на трех конкретных проблемах, возникающих при программировании процедуры *генмассив*.

Первая из них связана со спецификацией интерфейса процедуры. Ее заголовок должен выглядеть так:

функция *генмассив* (*n* : *цел*) : ↑ **массив** [...] **из** *t1*

Однако такая спецификация невозможна, так как тип *t1* статически неизвестен.

В некоторых языках класса С подобные проблемы решают путем введения так называемых «бестиповых» указателей. Тогда, например, можно дать такую спецификацию:

функция *генмассив* (*п* : *цел*) : ↑ *любой*

Здесь имеется в виду, что результатом процедуры является указатель объекта, тип которого может быть любым.

Использование бестиповых указателей существенно снижает надежность языка. Продемонстрируем это на примере. Пусть описана переменная такого же типа, что и результат *генмассив*:

перем *r* : ↑ *любой*

Предположим, что возможно присваивание:

r := *генмассив* (*k*)

Далее, присвоим *i*-му элементу массива какое-либо число:

r↑[*i*] := 100

Некоторые языки требуют, чтобы в подобных случаях точно указывался подразумеваемый тип указателя. Иными словами, надо выписать преобразование типа:

Пример 2.а.

изтип (*r* , ↑ **массив** [...] **из** *цел*) ↑ [*i*] := 100

Это позволяет, по крайней мере, внешне следовать принципу статического контроля типов. Используя то же средство, можно трактовать *i*-й элемент как указатель на объект типа целое. Тогда возможно присваивание:

Пример 2.б.

изтип (*r* , ↑ **массив** [...] **из** ↑ *цел*) ↑ [*i*] ↑ := 0

В результате число 100 будет трактоваться как указатель (или адрес) и соответствующая ячейка памяти будет обнулена. Таким способом можно получить доступ к произвольным участкам памяти, что и приводит к нарушению защиты.

Как видим, информация о типах, заданная в примерах 2.а и 2.б, не решает проблемы надежности. Она необходима транслятору, чтобы компилировать соответствующий код. Но ответственность за надежность этого кода лежит на программисте.

Вторая проблема возникает при описании базового массива. Его структуру можно схематически представить так:

Рис. 1.1. Структура массива *память*.

где С — свободные области, З — занятые, заштрихованные элементы — связующая информация списка.

Связующая информация имеет статически определенный тип. То же самое можно сказать и о свободных областях. Занятые области, строго говоря, имеют тип:

массив [...] **из** *t1*

Как уже упоминалось, тип *t1* процедуре неизвестен. Кроме того, у двух разных областей этот тип может быть различным. Учтем дополнительно, что области и связующая информация постоянно перемещаются по базовому массиву в процессе перераспределения памяти, изменяются в размерах, «переходят» одно в другое. Подытоживая, приходим к выводу, что трудно придумать такое статическое описание типа, которое в точности отображало бы логическую структуру этого массива (скорее можно сказать, что описанием структуры базового массива является собственно алгоритм процедуры *генмассив*).

По этим причинам, описывая базовый массив, указывают какой-либо «нейтральный» тип

перем ? *память* : **массив** [...] **из** *t2*

t2 может быть, например, типом «целое» или типом «связующая информация списков». Рассмотрим возможные последствия этой некорректности. Пусть задан тип «связующая информация»:

тип ? *сис* = **запись** *след* , *пред* : ↑ *сис* **конец**
перем ? *память* : **массив** [...] **из** *сис*

Программируя обработку списков («вплетение» в список и «выплетение» из списка), используют операторы такого вида:

память [*i*] . *след* := ↑ *память* [*j*]

(здесь в поле *след* *i*-го элемента заносится ссылка на *j*-й элемент). Предположим, фрагмент программы, выполняющий вычисление *j*, содержит ошибку. Тогда может оказаться, что при последующем проходе по списку в качестве его очередного элемента будет рассматриваться элемент какой-либо из областей (занятой или свободной); произвольная информация, содержащаяся в этом элементе, будет трактоваться как очередной элемент списка, т. е. как объект типа *сис*. На следующем шаге по списку, скорее всего, произойдет аналогичная ошибка и т. д.

Хотя процесс обработки списка пошел по неправильному пути, он может закончиться без единой диагностируемой ошибки. Однако логическая структура массива *память* будет искажена.

Последствия становятся наиболее ощутимы, если предположить, что *венмассив* — это системная процедура, а *память* — массив, отображающий всю оперативную память машины. В этом случае может оказаться, что будет испорчена информация, содержащаяся в тех участках памяти, которые должны быть наиболее защищены, а именно в таблицах операционной системы. Последнее обычно приводит к нарушению работы системы в целом.

Наиболее неприятные особенности подобных ошибок состоят в том, что а) от момента внесения ошибки до момента ее проявления может произойти достаточно много событий в системе, и б) характер проявления ошибки может не иметь ничего общего с ее изначальной причиной. И то, и другое серьезно затрудняет поиск и обнаружение ошибки.

Для дальнейшего изложения нам важно еще раз отметить, что указанные последствия являются результатом наложения двух факторов: ошибки в операторе программы и неверной трактовки информации. Последнее же вызвано тем, что, пытаясь остаться в рамках статики типов, мы при описании массива *память* придали ему смысл, недостаточно точно отражающий его реальную структуру.

Причина рассмотренных осложнений состоит в том, что языки класса С предлагают слишком ограниченную трактовку информационных структур. Ситуация стала бы значительно проще, если бы язык позволял описать следующий взгляд на данные, обрабатываемые процедурой *венмассив*:

1. Массив *память* — это массив элементов произвольного типа. Результатом процедуры является подмассив, выделенный внутри массива *память*. Подмассив состоит из *n* элементов (*n* — параметр процедуры). Поскольку тип не фиксируется, элементу подмассива в дальнейшем можно присваивать значение любого типа. С точки зрения алгоритма *венмассив* информация о типе тех значений, которые будут помещаться в подмассив, вообще не нужна. Важно лишь обеспечить, чтобы при последующем использовании подмассива нельзя было выйти за его границы. В противном случае будет возможно обращение к соседним участкам массива *память*, что приведет к нарушению защиты.

2. Тип текущего значения любого элемента массива *память* можно динамически анализировать. В нашем случае не нужно точно знать тип значения. Принципиально только проверить, является ли это значение объектом типа *сис*. Это позволяет достаточно просто организовать контроль. Например, если операции, связанные со списками, проверяют тип текущего обра-

батываемого элемента списка, то неверная трактовка информации исключается. Таким образом удается ликвидировать один из тех двух факторов, которые упоминались выше на стр. 34. Следовательно, ошибка в связующей информации обнаруживается еще до того, как она привела к серьезным последствиям. В этом смысле можно сказать, что динамика типов способствует раннему обнаружению достаточно сложных динамических ошибок.

Изложенный взгляд на структуру массива *память* достаточно просто описывается в языке класса Д. Однако средствами языка класса С нельзя достаточно адекватно отобразить эту модель. В результате возникают трудности, связанные со спецификацией процедуры *вемассив* и массива *память*.

Наиболее существенно, по-видимому, то, что ценой неадекватности является снижение надежности, а также снижение эффективности, поскольку приходится применять громоздкие методы контроля. Например, чтобы выяснить, содержит ли произвольно выбранный элемент массива связующую информацию списка, просматривается весь список, начиная с головы *). Это позволяет установить, является ли заданный элемент массива элементом списка.

Еще несколько слов о соотношении между статикой типов и диагностикой труднообнаруживаемых ошибок. Как известно, на поиск этих ошибок уходит основная доля времени, затрачиваемого на отладку. Однако, как видно, именно в этой области статический контроль типов не дает действенных результатов. Более того, здесь мы сталкиваемся с оборотной стороной статической концепции. Из нее следуют такие свойства языка, которые могут дополнительно усложнить диагностику труднообнаруживаемых ошибок.

Пример 3. В качестве последнего примера рассмотрим одну из проблем, возникающих при реализации диалоговой системы с помощью ЯВУ.

Проблема состоит в следующем: какими средствами кодировать вызов программы, алфавитно-цифровое имя и параметры которой поступают с терминала? Языки класса С устроены таким образом, что обозначение каждой вызываемой процедуры, спецификация типов параметров и результаты известны во время трансляции. В данном случае это требование не выполняется, так как алфавитно-цифровое имя программы возни-

*) См. метод контроля, применяемый в одном из возможных алгоритмов распределения памяти — «алгоритме близнецов» [28].

кает динамически; все связанные с нею сведения также могут быть установлены только динамически. В результате прямыми средствами языка невозможно выразить такой вызов. Необходимо использовать какие-либо косвенные средства: выход в код или обращение к специальной системной подпрограмме; последняя в свою очередь должна воспользоваться средствами, выходящими за рамки возможностей языка класса С.

* * *

Приведенные примеры связаны одной общей характеристикой: в каждом случае рассматривалась ситуация, для которой характерна динамика типов обрабатываемых объектов. Именно в этих случаях проявляются недостатки прямой связи имени с типом и косвенной связи объекта с его типом через посредство имени.

Дополнительно отметим недостаток методологического характера. В ряде случаев в исходном алгоритме не требуется полная информация о типах. Такая ситуация имеет место и в рассмотренных примерах. Однако, ввиду того, что имя должно быть связано с каким-либо типом, необходимо вводить в программу фиктивные типы, которые не имеют реального смысла и усложняют описание алгоритма.

Причины введения статики типов. Семантический уровень традиционных машин ниже уровня семантической модели языка программирования. В частности, в памяти машины хранятся данные единственного типа — последовательность битов. Наличие только одного типа равносильно тому, что для памяти вообще отсутствует понятие тип. Смысловая трактовка двоичной информации возникает тогда, когда над ней выполняется операция. Причем различные операции могут приписывать различный смысл одной и той же последовательности битов. Поэтому можно сказать, что машинный язык — это бестиповый язык. В этом наиболее существенное отличие машинного языка от ЯВУ.

Причины статики типов можно просто объяснить, если проанализировать процесс поэтапного «перерастания» машинного языка в ЯВУ. При этом мы будем учитывать соображения эффективности реализации конечного ЯВУ.

Бестиповый язык. На первом этапе сконструируем язык, схожий с ЯВУ по форме операторов, выражений, структур управления (условные операторы, циклы, процедурный механизм) и пр. Исключение состоит в одном: этот язык бестиповый. Например, в нем есть описание переменных:

переменная x , y

Для каждой переменной отводится одно слово памяти. Тип возможных значений не указывается.

Описание массива может иметь вид:

переменная *a* [100] , *b* [10] ;

В первом случае отводится 100 слов памяти, а во втором — 10.

Переменной можно присвоить какой-либо объект, например, вещественное число:

$x := 3.14$

но информация о типе присвоенного значения нигде не сохраняется. Чтобы транслятор мог различать операции целой и вещественной арифметики, их надо по-разному обозначить. Например,

x* плюсв *y

означает сложение вещественных, а

i* плюсц *j

сложение целых. Таким образом, пришлось отступить от принципа универсальности операций.

Универсальность операций можно понимать шире. Например, будем считать, что конструкция «переменная с индексом»

$a[i]$

есть применение операции индексации к двум операндам. Универсальность этой операции состоит в том, что ее обозначение не зависит от типа элементов индексируемого массива. Например, для элемента массива вещественных чисел обычно отводится одно слово, а для элемента массива логических значений — один разряд. Следовательно, индексацию в этих двух случаях надо транслировать в разные последовательности команд. Однако в ЯВУ она обозначается одинаково.

В случае бестипового языка придется и здесь отказаться от универсальности обозначений. Например, если с точки зрения алгоритма массив *a* — это словный вектор, а *b* — битовый, то индексация первого обозначается обычным образом

$a[i]$

а индексация второго несколько иначе:

***b* бит** [*i*]

Только тогда транслятору будет известно, какую последовательность команд генерировать в каждом случае.

Итак, первое отличие сконструированного языка от ЯВУ заключается в том, что его операции не обладают свойством уни-

версальности. Второе отличие — в степени надежности. Уже упоминалось, что в бестиповом языке отсутствует контроль за правильностью смысловой трактовки информации. Это приводит не только к ошибкам в арифметических и логических операциях. Более серьезные последствия возникают при неправильной работе с адресной информацией. Например, в бестиповом языке любое целое может быть случайно использовано как указатель массива. При этом, в лучшем случае, может произойти нарушение межпроцедурной защиты, приводящее к таким трудно обнаруживаемым ошибкам, как запись в логически недоступную переменную или массив. В худшем случае возможно нарушение защиты между пользователем и системой, что может привести к нарушению работы всей системы в целом.

Язык с системой типов. На первом этапе мы, хотя и приблизились к ЯВУ, но еще не достигли достаточного сходства. Осталось ввести в язык средства, обеспечивающие универсальность операций и надежность. И то и другое можно достичь, введя механизм типов. Вообще говоря, понятие «тип» уже неявно подразумевалось нами и выразилось в номенклатуре операций языка. Теперь это надо сделать какими-то явными средствами.

Существуют два основных подхода. Первый из них состоит в следующем. В информационном содержимом объекта некоторым образом кодируется его тип. Например, для кодировки типа отводится несколько стандартных разрядов. Универсальная операция транслируется в такую последовательность команд, которая производит анализ типа операндов и выбирает нужный алгоритм выполнения операции. Этот процесс можно назвать динамической идентификацией операций. Как видно, одновременно с динамической идентификацией можно проконтролировать допустимость типов. Таким образом, неверная трактовка информации исключается, т. е. обеспечена не только универсальность, но и надежность.

Этот подход применяется в языках класса Д [9, 10, 13—16]. Недостаток такого подхода в случае традиционных машин — это интерпретационное выполнение операций. Следовательно, исходное требование эффективной реализации не удовлетворяется. Поэтому мы воспользуемся другим подходом — статическим. Существо статического подхода легко иллюстрируется на примере переменных. Тип связывается не со значением переменной, а с ее обозначением, с идентификатором *). Эта

*) Формально это делается опосредствованным образом. Обозначению, связанному с именем, приписывается тип этого имени. Тип имени однозначно задает тип именуемого объекта.

связь задается статически, в тексте программы. Транслируя универсальную операцию, транслятор по идентификаторам операндов делает заключение о типе значений и генерирует код соответствующей конкретной операции. Этот прием называется статической идентификацией операций. Он возможен лишь в том случае, если обозначения связаны со статически фиксированными типами. Это же позволяет во время трансляции осуществить контроль типов и тем самым исключить неверную смысловую трактовку информации.

Таким образом, требование обеспечить эффективную реализацию таких важных свойств ЯВУ, как универсальность операций и надежность, в условиях традиционных машин приводит к статике типов. Как видим, этот прием имеет вполне объективную основу, чем и объясняется его применение в большинстве распространенных языков программирования.

Ограничения, возникающие из-за статики типов, можно преодолеть путем введения в язык средств такого рода, как показанные выше в примере 2. При этом, естественно, теряется надежность. Действительно, применяя бестиповые указатели и неконтролируемое преобразование типов, мы в сущности возвращаемся на шаг назад, к бестиповому языку. Тот факт, что в тексте программы фигурирует информация о типах, роли не играет. Например, запись (см. примеры 2.а и 2.б)

изтип (*а*, массив [...] из *вещ*) [*i*]

практически эквивалентна особому обозначению операции индексации массива вещественных чисел. Оно используется транслятором, чтобы генерировать соответствующую последовательность команд, но ничего не дает с точки зрения надежности.

Доказательство правильности программы. Статическое описание типа имеет еще одно применение. Его можно рассматривать как информацию об абстрактных свойствах именуемого объекта. Тот факт, что эти свойства указываются в описании, говорит о том, что они являются инвариантными относительно процесса выполнения программы. В соответствии с этим статические языки формулируются таким образом, что правильная с точки зрения языка программа не может содержать операторы, нарушающие эти инварианты. Транслятор, осуществляя синтаксический и семантический контроль, по существу конт-

---

та. Таким образом, по обозначению можно установить тип именуемого объекта. Этим и обусловлен тот факт, что имя статически связано с типом именуемых значений.

ролирует сохранение инвариантов. Ошибки, которые могли бы вызвать изменение статических свойств объектов, обнаруживаются при трансляции.

В настоящее время определился класс языков с полным статическим контролем типов: алгол-68, clu, modula, euclid, tartan, ada [7, 17—21]. Однако для описания связей в сложном динамическом случае необходимо, чтобы возможности языка в части высказывания утверждений об инвариантных свойствах объектов были существенно шире, чем те, которые есть в этих языках.

Последовательное развитие идеи инвариантов приводит к языку спецификаций. Спецификации используются двумя существенно различными способами. Первый из них использует спецификации для затруднения написания неправильных программ, а второй — для облегчения написания правильных. В первом направлении наиболее продвинуты работы по языку alphard [5, 22]. Предоставляемый этим языком аппарат высказываний представляет безусловный теоретический интерес. Трудность такого подхода, как отмечается, состоит в том, что навязываемая программисту синтаксическая сложность противоречит цели облегчения программирования. Большой процент текста программ на языке alphard представляет собой избыточную информацию, единственная цель которой — формальное доказательство правильности программ (см., например, описание и доказательство формы «таблица символов» [5], реализующей довольно простые функции сканера в трансляторе).

Второй способ использует спецификации для синтеза программ. Примером такого подхода является язык сетл [6]. Практическое применение этого языка показало, что необходимо дополнительное средство, позволяющее повысить эффективность синтезируемой программы. С этой целью был разобран подъязык представлений, предназначенный для описания реализации абстрактных понятий. Интересно отметить, что подъязык основан на динамике типов. Хотя он и допускает спецификацию типа переменной, но, как отмечают авторы сетла, эти спецификации используются исключительно для повышения эффективности кода. Таким образом, динамика типов и статическое описание инвариантов не противоречат, а наоборот, дополняют друг друга.

Для данного анализа важно отметить, что не следует переоценивать значения статики типов в области доказательства правильности программ. Пример языка alphard показывает, что статика типов — это лишь одна из возможных разновидностей инвариантных соотношений.

Язык с полным статическим контролем типов не исключает необходимости явно программируемого динамического контроля, причем на практике может оказаться, что этот контроль значительно менее эффективен, чем в динамическом языке. Пример подобной ситуации был продемонстрирован выше при разборе обстоятельств, связанных с программированием процедуры распределения памяти (см. стр. 35).

Средства ограниченной динамики. Недостатки строгой статики типов частично ликвидируются путем введения в язык средств, которые можно было бы охарактеризовать как средства ограниченной динамики. Сюда относятся объединенные типы языков алгол-68, clu и mary [23], контролируемые вариантные записи языков euclid и ada. Подробный анализ методов выходит за рамки книги. Здесь отметим лишь три обстоятельства.

1. Повсеместное введение этих средств является признанием необходимости уменьшить консервативность распространенных языков в части управления типами.

2. Трактовка динамики типов в языках класса С имеет определенную специфику, приводящую к некоторым неудобствам при программировании. Чтобы ее пояснить, вернемся к примеру 1.б. Возникшее там увеличение текста программы обусловлено следующим.

При введении динамики типов в язык класса С надо каким-то образом примирить две противоречивые посылки. С одной стороны, надо допустить некоторую неопределенность в типах (в примере 1.б тип элемента массива *в* может быть либо целым, либо вещественным). С другой стороны, учитывая соображения эффективности, трансляцию универсальных операций надо проводить по методу статической идентификации. Но статическая идентификация таких стандартных операций, как арифметические, индексация и ряд других, основана на том, что тип операндов статически определен и не является объединением из нескольких возможных типов. Противоречие разрешается следующим образом. Вводится конструкция, называемая «сопоставляющее предложение» *) (в примере 1.б — правая часть оператора присваивания в цикле). Она позволяет запрограммировать динамический перебор возможных вариантов и свести объединенный тип (цв) к его составляющим (цел или вещ). Это дает возможность в конечном счете выписать операцию в

*) Такое название используется в алголе-68. В языках euclid и ada схожая конструкция называется дискриминантным предложением.

таком виде, при котором возможна статическая идентификация (как видно из примера, при трансляции каждого из четырех вариантов выражения $c + вк$ известны точные типы c и $вк$).

Применив сопоставляющее предложение, мы, в сущности, явным образом запрограммировали процесс динамической идентификации операций. Этого и следовало ожидать. Проверки, связанные с динамикой типов, имеют объективный характер; в процессе выполнения программы их надо тем или иным образом выполнить. В языке класса Д они неявным образом выполняются универсальной операцией. В языке класса С их надо программировать явным образом.

Тем самым как бы выражен следующий принцип конструирования наиболее последовательных языков класса С: Язык гарантирует эффективность выполнения конструкций, основывающихся на статике типов. Что же касается динамики, то язык не скрывает того факта, что здесь возможны потери в эффективности. Все проверки, связанные с динамикой типов, должны быть запрограммированы явно, чтобы по тексту программы можно было бы сделать заключение о возникающих при этом накладных расходах. Такой подход отображает ориентацию языка на класс машин с бестиповым машинным языком. Как видим, избыточность текста программ оправдана прагматическими соображениями.

3. Наконец, третье обстоятельство состоит в том, что методы ограниченной динамики не решают проблем программирования алгоритмов, существенно динамичных по типам обрабатываемых данных. Чтобы убедиться в этом, достаточно рассмотреть примеры 2 и 3. В каждом из этих случаев оказывается, что ни один из методов не дает принципиально новых возможностей. Основная причина в том, что степень свободы в части типов ограничена: имя может быть связано с некоторым набором типов, но этот набор — ограниченный и по-прежнему статически определяемый.

Заключительное замечание. Подводя итог, рассмотрим вопрос о возможном балансе между статическими и динамическими методами контроля. Отметим, во-первых, что в ряде случаев статически известны некоторые соотношения между объектами программы, сохраняющиеся в процессе ее выполнения. Подобные инварианты имеет смысл контролировать во время трансляции. Одной из частных разновидностей такого контроля является статический контроль типов.

С другой стороны, для корректного решения задач, подобных тем, что упоминались выше, требуется гибкость языков

класса Д. Поэтому, по-видимому, целесообразно в качестве языка-ядра рассматривать язык, динамический в части типов, и иметь дополнительно средства, позволяющие при необходимости описать и проконтролировать те статические соотношения, которые свойственны природе данной программы. В частности, такой подход позволяет в случае надобности задать статический контроль типов.

Здесь надо подчеркнуть, что в качестве основы выбирается язык класса Д, а над ним уже делается статическая надстройка. В упоминавшихся выше языках класса С предпринят по объективным причинам иной путь, т. е. динамика типов строится на базе статики типов. Это и приводит к недостаточно удовлетворительным результатам.

3.2. Генерация объектов. В языках высокого уровня существует ряд ограничений на степень динамики в части создания и уничтожения объектов (в этом разделе под термином «объекты» подразумеваются оперативные объекты, т. е. такие, создание которых связано с отведением оперативной памяти, например имена и массивы). Фортран является языком с полностью статическим распределением памяти; все объекты создаются в начале выполнения программы. В алголе длина массивов вычисляется динамически, но акты создания и уничтожения объектов жестко связаны с фразовой структурой программы. В языках, последовавших за алголом (симула-68, алгол-68, паскаль), были введены средства динамической генерации объектов, т. е. объект может быть создан в произвольный момент (см. использование процедуры *нов* в § 2). Время жизни вновь созданного объекта определяется в разных языках по-разному. В основном применяется один из двух подходов:

1) время жизни объекта не связывается с блочной структурой программы, но предоставляются средства явного уничтожения объектов;

2) при создании объекта можно указать, уничтожается ли он автоматически по завершении выполнения ближайшего охватывающего блока, либо же существует до конца выполнения программы.

Степень динамики в области генерации объектов тесно связана с вопросами реализации. Языки со статическим распределением памяти могут быть просто реализованы в рамках операционной системы, обеспечивающей строго статическое распределение ресурсов. Схема распределения памяти языка алгол-60 может быть осуществлена по принципу «магазина» (стека). Такая реализация также не предъявляет существенных требований к операционной системе. В отличие от этого,

динамическая генерация объектов предполагает наличие средств динамического распределения ресурсов. Слабая поддержка динамической стратегии со стороны аппаратуры и операционной системы приводит к значительным накладным расходам на использование этого механизма. Эти расходы выражаются и в замедлении программы, и в увеличении требуемого ею пространства памяти.

По этой причине современные языки высокого уровня придерживаются двойственной позиции. С одной стороны, в языке есть статические описания объектов, подразумевающие магазинно-ориентированный режим распределения памяти, с другой стороны — существуют динамические генераторы. В практике программирования генераторы, по возможности, не используются, чтобы не предпринимать в программе специальных мер для снижения связанных с ними накладных расходов. В основном используются статические описания. Недостатком здесь является то, что в ряде случаев это приводит к необходимости принципиального изменения исходного алгоритма и представления данных.

3.3. Обособление внешних объектов. Внешние объекты — это объекты, располагающиеся на внешних носителях. В первую очередь к ним относятся такие традиционные объекты алгоритмических языков, как файлы. Реально в вычислительной системе существуют и другие внешние объекты, например архивные справочники, семейства пакетов дисков*) (или бобин магнитных лент). Наконец, программа — это тоже внешний объект, создаваемый транслятором в результате трансляции файла текста программы.

В отличие от объектов, располагающихся в оперативной памяти (например, массивов), время жизни внешних объектов не ограничивается временем выполнения программы. Такой объект может существовать до начала выполнения программы и продолжать существовать по окончании выполнения. Если в вычислительной системе реализован архив, то с внешним объектом можно некоторым образом связать алфавитно-цифровое обозначение — «внешнее обозначение». Оно используется для того, чтобы идентифицировать внешний объект. Совокупность внешних обозначений можно, по аналогии с контекстом обозначений, описанных в программе, назвать контекстом внешних обозначений. Время жизни внешнего контекста, как и вре-

---

*) Например, в ряде систем несколько пакетов дисков можно объединить в семейство и дать семейству некоторое алфавитно-цифровое обозначение.

мя жизни внешних объектов в общем случае не ограничивается временем выполнения программ.

Теперь обратимся к языкам высокого уровня. Существует определенная специфика в том, как в них трактуются внешние объекты:

1) Средствами языка можно управлять только файлами. Другие разновидности внешних объектов недоступны в программах. Их создание, обработка и уничтожение описываются вне программы, например в пакетном задании, которое программируется на специальном языке (так называемом языке управления заданиями).

2) ЯВУ не определяют исчерпывающим образом, как строится контекст внешних обозначений, каковы правила поиска объектов по их обозначениям. Некоторые неформальные средства существуют. Например, файлы алгола-68 имеют атрибут *обозначение*, содержащий строку литер. Подразумевается, что эта строка и есть внешнее обозначение. Однако дальнейшее остается неопределенным. В частности, пусть одна программа связала с файлом строку *s*, а другая программа того же пользователя открывает файл, указывая при этом, что атрибут *обозначение* равен *s*. Будет ли это один и тот же файл? Может ли программа другого пользователя, используя аналогичным образом строку *s*, открыть тот же файл? Определение языка не дает ответ на этот вопрос.

3) Средства управления файлами внесены в язык лишь частично. Наиболее полно представлены операции обмена, тогда как семантика создания, именования и уничтожения либо не определена вообще, либо имеет значительное число неопределенных мест. По этой причине генерация файлов, как правило, задается вне программы; в программе описываются только некоторые логические характеристики уже существующего объекта. Внешние обозначения обрабатываемых файлов также указываются не в программе, а на уровне задания.

В этом состоит обособленность внешних объектов в языках высокого уровня: не все действия, связанные с такими объектами, можно описать на языке. Причины перечисленных ограничений будут рассмотрены ниже. Здесь остановимся на нескольких примерах, иллюстрирующих недостатки такого подхода.

Примеры. В качестве первого примера рассмотрим потребности, возникающие при программировании трансляторов. Сначала несколько слов о выходном файле объектного кода. Дело в том, что транслируемая программа может содержать такие ошибки, которые делают нецелесообразной генерацию этого

файла. Поэтому естественно, чтобы решение о создании файла принималось не в программе задания, а в процессе работы транслятора. Если генерация кода прошла успешно, то файл возвращается как результат трансляции; в противном случае его надо уничтожить до завершения трансляции.

В отношении рабочих файлов многопроходного транслятора можно привести аналогичные соображения. Эти файлы являются чисто внутренними объектами транслятора; их создание следует программировать на соответствующих фазах трансляции, но не в программе, выполняющей вызов транслятора.

Таким образом, первый недостаток перечисленных ограничений состоит в том, что они приводят к немодульному программированию: часть информации о локальных объектах приходится задавать вне ее. Следующие примеры касаются способов именования файлов.

При трансляции операторов типа include (PL/1) и copy (кобол) необходимо открывать файлы, внешние обозначения которых поступают из текста транслируемой программы, т. е. вычисляются динамически. Еще большие требования возникают при программировании диалоговой системы. Существенная особенность подобной системы состоит в том, что она создает новые внешние объекты, узнает из входных приказов обозначения существующих объектов, открывает их, изучает атрибуты, осуществляет обмен с файлами и вызов программ. Таким образом, здесь наблюдается принципиально динамическая картина взаимодействия с внешними объектами, когда статически неизвестно число обрабатываемых объектов, их внешние обозначения и атрибуты. Выделение внешних объектов в особый класс объектов, создание и именование которых осуществляется неязыковыми средствами, усложняет программирование подобных алгоритмов на языке высокого уровня. В этих случаях, как правило, используется язык ассемблера.

Как уже отмечалось, особая трактовка внешних объектов состоит и в том, что семантика ЯВУ не описывает законов построения контекста внешних обозначений; способ поиска объекта по обозначению зависит от используемой операционной среды. Вследствие этого смысл внешних обозначений существенно отличается от смысла обозначений, описываемых в программе. Если для вторых используется принцип статической известности, то для первых применяется механизм динамической известности, что может явиться причиной нарушения межпрограммной защиты. Пусть, например, в программе описан файл:

файл *f*;

В некоторых системах соответствие между внешним обозначением файла и обозначением, связанным с файлом при описании его внутри программы (в нашем случае *f*), определяется на уровне задания. В результате, если в двух независимых программах описаны различные файлы с одинаковым программным обозначением, то при вызове одной программы из другой произойдет обращение к одному и тому же файлу, а именно — к файлу, определенному в задании.

Причины обособления внешних объектов. Реализация таких объектов, как программа, файл, архивный справочник, подразумевает решение широкого круга вопросов: загрузка программ, организация связи программы с внешними объектами, распределение вторичной памяти, организация обмена и обработки ошибок, реализация архива. Эти вопросы, в свою очередь, связаны с распределением оперативной памяти и управлением процессами. Как видно, перечисление охватывает весь спектр задач, решаемых операционной системой. В связи с этим взаимодействие с внешними объектами осуществляется средствами операционной системы.

С другой стороны, семантический уровень традиционных операционных систем значительно отличается от уровня развитых языков программирования. Причиной этому является то, что операционные системы, в первую очередь, разрабатывались с целью повышения производительности машины и не были ориентированы на проблемный уровень. При таком положении целесообразно вывести внешние объекты из языка, оставив лишь простейшие конструкции, связанные с открытием и обменом. Все остальное взаимодействие описывается на так называемом языке управления операционной системой (вариант этого языка, обеспечивающий пакетный режим, называется языком управления заданиями). Таким образом, причина обособления внешних объектов — это низкий семантический уровень понятий, реализованных в операционной системе.

Язык управления заданиями. Наличие дополнительного языка является еще одним недостатком традиционного подхода. Для реальной работы в системе программисту необходимо изучить, по крайней мере, два языка — проблемный язык и язык управления заданиями. Важно отметить, что эти языки существенно различаются по семантическим понятиям и синтаксическим формам. В этом отношении наглядным примером служит язык JCL, используемый в системах IBM 360/370 для управления заданиями. В начале 70-х годов наметилась тенденция к повышению уровня языков управления операционной системой (см., например, [24—26]). Однако доступность подоб-

ного языка не решает проблему полностью, так как с точки зрения практика программирования удобно было бы иметь единый язык высокого уровня, позволяющий, в качестве одного из частных применений, описывать алгоритмы пакетных заданий.

3.4. Ограниченная рекурсивность языка. В § 2 уже отмечалось, что алгоритм, получающийся в результате пошаговой конкретизации абстрактных действий, имеет рекурсивную фразовую структуру. Можно, кроме того, сказать, что с рекурсивной фразовой структурой тесно связано и понятие «модульность», поскольку оно также подразумевает возможность независимо описывать алгоритмы абстрактных операторов.

Адекватное отображение рекурсивной фразовой структуры алгоритма возможно в том случае, если язык определен полностью рекурсивно. Это означает, что на место абстрактного оператора может быть подставлено описание алгоритма произвольной сложности, реализующего этот оператор. Аналогично рекурсивность в части выражений означает, что на место абстрактного операнда любого выражения может быть подставлено сколь угодно сложное выражение, выполняющее соответствующее вычисление.

В распространенных языках (фортран, алгол, паскаль) накладывается ряд ограничений на рекурсивность синтаксических форм. Существо ограничений состоит в том, что операнды некоторых конструкций могут быть представлены только в форме простейших выражений. Например, в конструкции «элемент массива» может быть использован идентификатор массива, но не условное выражение, выбирающее один из двух массивов, и не вызов процедуры, поставляющей нужный массив. Аналогичным образом ограничена синтаксическая форма присваивания, вызова процедуры и ряда других конструкций.

В языке паскаль и его преемниках введены дополнительные ограничения. Здесь отсутствуют условные арифметические и логические выражения и многократные присваивания. В целом это означает, что свойственная алгоритму рекурсивная вложенность действий может быть отображена только на уровне операторов. Например, присваивание:

если q **то** x **иначе** y **все** := e

необходимо отображать в виде условного оператора с двумя присваиваниями:

если q **то** $x := e$ **иначе** $y := e$ **все**

Другой пример. Индексацию:

$\ldots p\ (x, y)\ [k] \ldots$

можно выполнить только путем введения дополнительной переменной:

$a := p\ (x,\ y)$;

$\ldots a\ [k] \ldots$

Существует несколько причин, по которым ограничивается рекурсивность языка. Первая заключается в том, что в языке со статическим контролем типов возникает проблема балансировки типов альтернатив в условных и выбирающих предложениях. Действительно, если операндом какой-либо универсальной операции является альтернативное предложение, то чтобы осуществить статическую идентификацию операции, нужно все альтернативы привести к одному типу. Например, рассмотрим выражение:

если q **то** x **иначе** k **все** $+\ i$,

где k и i — типа целое, а x — типа вещественное. Сначала надо преобразовать k к типу вещественное. Это позволит считать, что результатом выполнения условного предложения всегда является вещественное число. Только после этого в качестве конкретной операции можно выбрать такую операцию сложения, у которой левый операнд вещественный, а правый — целый. Этот простейший пример демонстрирует природу балансировки типов. Отметим, что проблема не ограничивается лишь преобразованиями целое — вещественное. Существуют и более сложные примеры, обсуждение которых выходит за рамки книги.

Если в языке кроме того допускается «перегрузка» операций (описание одной операции для различных сочетаний типов операндов), то могут появиться двусмысленности. В связи с этим необходимо либо ограничивать рекурсивность, как в языке ada, либо вводить нетривиальные правила балансировки типов операндов перегруженных операций, как в языке алгол-68. Похожие проблемы преобразования типов возникают и в том случае, если в статическом языке разрешена рекурсия в присваивании и в некоторых других конструкциях.

Другой причиной ограничений являются соображения, связанные с упрощением трансляции. Дело в том, что из-за различия уровней языка и системы команд возникает ряд трудностей при генерации кода рекурсивных конструкций: проблема распределения явно адресуемых регистров, генерация кода записи в статически неизвестную часть слова (случай рекурсии в левой части оператора присваивания), генерация эффективного кода динамического контроля, например, кода конт-

роля границ массива и т. д. Если язык рекурсивен только в области арифметических и логических выражений, то часть проблем ликвидируется.

3.5. Структуры управления. Соответствие между фразовой структурой текста программы и структурой динамического процесса ее выполнения достигается в том случае, если в программе используется лишь ограниченный набор средств передачи управления: прямая текстуальная последовательность действий, выбор одного из альтернативных действий, циклическое выполнение действия и вызов процедуры. В настоящее время соответствующие конструкции достаточно устоялись и существуют во многих языках высокого уровня. Однако такая номенклатура структур управления не позволяет полностью исключить из языка оператор перехода и связанные с ним неограниченные возможности передачи управления.

Существует ряд особых обстоятельств, для обработки которых недостаточно перечисленных структур управления. К числу таких обстоятельств относятся:

— ошибки, приводящие к возникновению аварийной обстановки. В ряде случаев требуется, чтобы в программе были предусмотрены алгоритмы обработки таких ошибок;

— обстоятельства, ошибочные с точки зрения процедуры, которая их обнаруживает, но являющиеся нормальными для процедуры, вызывающей первую. Типичный пример — это попытка чтения за пределами заполненной части файла (так называемая ошибка «логический конец файла»). Процедура чтения квалифицирует такую попытку как ошибочную. Однако для программы, читающей файл, это — заранее предусмотренное обстоятельство, возникновение которого говорит об исчерпании файла;

— выполнение условия окончания цикла. Здесь мы предполагаем, что проверка условия возможна в любой точке внутри цикла. При выполнении условия цикл прекращается, а операторы, оставшиеся невыполненными на последней итерации, пропускаются;

— обстоятельства, приводящие к прекращению выполнения процедуры еще до достижения ее текстуального конца и т. д.

Во всех этих случаях в языке нужно иметь средство для прекращения нормального процесса выполнения текущей и, возможно, нескольких охватывающих фраз. Поскольку такого специального средства в распространенных ЯВУ нет, то передача управления осуществляется с помощью оператора перехода. Данные, характеризующие возникшие обстоятельства, передаются через глобальные переменные. Иногда дополнитель-

но используется некоторое «особое» значение, которое поставляется процедурой при возникновении внутри нее ошибочной ситуации. Это значение обрабатывается в месте вызова, и далее процесс аварийного возврата может повториться. Обработку особых обстоятельств можно программировать и без оператора перехода, например, с использованием условных операторов, охватывающих часть процедуры от места возникновения аварийной ситуации до ее текстуального конца. В целом такие методы представляются искусственными и затрудняющими отладку и изучение программ. Для решения этой проблемы набор структур управления должен содержать конструкцию, адекватно отображающую действия, связанные с обработкой особых обстоятельств. Это позволит полностью исключить из практики программирования необходимость в использовании оператора перехода.

3.6. Расслоение языков высокого уровня. Выше были обсуждены такие черты распространенных ЯВУ, которые в ряде случаев затрудняют их использование. В каждом отдельном случае отмечалось, что причиной этого является низкий семантический уровень машинного языка и/или примитивов операционной системы. Подытоживая, рассмотрим соотношение между уровнями понятий исходного алгоритма *), языка, операционной системы и машины. Условно это соотношение можно представить следующим образом:

понятия алгоритма	
понятия языка	проблемный язык
понятия операционной системы	язык управления операционной системы
понятия машины	машинно-ориентированный язык

*) Несколько слов о термине «понятия исходного алгоритма». Используемый язык программирования, безусловно, оказывает влияние на способ мышления программиста, на то, как он представляет исходный алгоритм до воплощения его в программу на языке. Надо тем не менее признать, что категории, используемые при формулировке исходного алгоритма, часто имеют более высокий уровень, чем языковые понятия. Например, описывая исходный алгоритм в п. 3.1 (см. рис. 1.1), мы графически изобразили одну структуру данных, а в программе, из-за ограниченности средств языка, описали другую — более регулярную и простую.

Компромисс между уровнем алгоритма и двумя низшими уровнями достигается путем введения промежуточного языкового уровня. В свою очередь уровень языка выбирается таким образом, чтобы не привести к значительным потерям эффективности при отображении исходного алгоритма в понятия низших уровней.

Схема соотношения уровней объясняет происхождение рассмотренных ранее проблем применения ЯВУ:

— понижение уровня языка ограничивает его возможности. Оказывается, что ряд алгоритмов в принципе невозможно описать, строго придерживаясь исходных концепций языка. В особенности это относится к алгоритмам, для которых характерны динамика типов и интенсивное взаимодействие с внешними объектами. В подобных случаях используются средства ассемблерного программирования;

— существование различия между уровнем языка и уровнем машины говорит о том, что проблема эффективности кода полностью не решена. Поэтому в ряде случаев даже при описании вычислительных алгоритмов используются ассемблерные средства;

— на уровне машины существует ряд специфических механизмов, например, особенности адресации данных и кода, особенности системы обработки прерываний. Чтобы эффективно использовать возможности конкретной аппаратуры, системные алгоритмы обычно учитывают подобные механизмы. Поскольку для них нет прямых аналогов в ЯВУ, программирование ведется с помощью специальных ассемблерных средств.

Современные языки высокого уровня, предназначенные для использования в области системного программирования, по существу являются многоуровневыми *). Наряду с универсальными конструкциями они содержат и сугубо системные возможности. Формы, в которых представлены эти возможности, существенно различаются:

— явное использование регистров (GSL [28], PL/1 [28], GPL [28], chili [28]);

— неконтролируемое преобразование типов (симула-67, mary, euclid, ada);

— бестиповые указатели (LIS [29], PL/1, GSL, PS 440 [30]);

— арифметика над указателями (chili, bliss [31]);

---

*) Иногда (как, например, в системе Burroughs [27]) вводится несколько языков, различающихся по возможности доступа к системным и аппаратным средствам.

— неконтролируемые вариантные записи (паскаль*), sydec [28]);

— машинно-ориентированные модули (euclid);

— спецификация реализации (ada, chili).

В целом эти средства можно охарактеризовать как ассемблерные в том смысле, что они допускают безконтекстную обработку информации, т. е. обработку, не связанную с типом данных и доступным пространством имен. Как видим, в каждом отдельном случае расширение возможностей достигается, во-первых, за счет уменьшения надежности языка и, во-вторых, посредством его расслоения, т. е. путем снижения степени целостности и концептуального единства языка. Уменьшение надежности следует признать основным недостатком такого подхода.

4. Выводы. Характеристики автокода

На основе проведенного анализа можно сделать вывод о том, что задача развития средств программирования предполагает повышение семантического уровня машинного языка и примитивов операционной системы. Это позволяет, во-первых, внедрить новые ЯВУ, свободные от ограничений, вызванных несовершенством среды реализации, и во-вторых, увеличить эффективность, а следовательно и сферу применения существующих ЯВУ.

В § 1 уже отмечалось, что очевидным следствием такого подхода является принятая в системе Эльбрус методика совместной разработки системы команд, математического обеспечения и базового языка программирования, которым является автокод эльбрус. Еще раз подчеркнем, что задачей совместной разработки являлось не создание аппаратного семантического процессора для некоторого конкретного языка высокого уровня, а выявление, развитие и аппаратная реализация совокупности фундаментальных понятий процессов обработки информации.

Автокод характеризуют следующие свойства: динамика типов, динамическая генерация объектов, наличие средств непосредственного взаимодействия с внешними объектами, полная рекурсивность, наличие развитых структур управления, включая средства обработки особых обстоятельств.

---

*) В описании языка паскаль не определено, осуществляется ли динамический контроль типа вариантной записи. Что же касается реализаций, то в большинстве из них такой контроль не проводится.

Перейдем к подробному обсуждению этих характеристик. При этом будем проводить параллель между понятиями языка и понятиями системы. Чтобы избежать повторов, в тех случаях, когда свойства языка и системы полностью совпадают, будем говорить либо только о языке, либо только о системе.

4.1. Общий подход. Начнем с краткого обзора особенностей автокода. Для этого рассмотрим изображенную на рис. 1.2 картину взаимосвязи понятий языка. Поясним этот рисунок. Процедура (в частности, операция) связана с доступным ей контекстом обозначений. Элемент контекста обозначений связан с

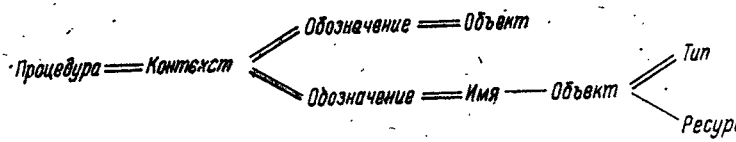


Рис. 1.2. Общая схема понятий.

объектом (на рис. изображены два таких элемента). Если объект не является именем, то такая пара называется константой (верхняя пара). В противном случае пара образует переменную (нижняя пара). Имя именует некоторый объект. С объектом связан тип и ресурс, занимаемый объектом.

Связь имени с объектом является переменной, т. е. объект, находящийся на конце связи, может заменяться на другой объект. Связь между объектом и ресурсом в общем случае переменная. Эта связь переменная в том смысле, что требуемый ресурс может меняться в размере. Например, можно увеличивать размер массива или файла. Остальные связи — фиксированные; они выделены двойной линией.

Рассматриваемая картина связей имеет ряд важных последствий:

а) Во-первых, отметим, что имя непосредственно связано не с типом, а с объектом. Одно и то же имя в разное время может быть связано с объектами различных типов; тип текущего значения переменной можно анализировать, и в зависимости от этого предпринимать те или иные действия. Это свойство называется динамикой типов.

б) Возможен случай, когда в контексте процедуры известен не весь набор операций над объектами некоторого типа, а только, так сказать, «безопасные операции», не приводящие к нежелательным последствиям. Это свойство обеспечивает защиту данных (подробнее см. п. 4.6).

в) Ресурс связан с именем косвенно, через понятие объект. Опосредованность этой связи означает, что алгоритм формули-

руется в терминах объектов и, следовательно, не описывает непосредственно распределение физических ресурсов.

г) Наличие переменной связи между обозначением и объектом (косвенно, через имя) означает, в частности, что акты создания и ликвидации объекта могут быть не связаны непосредственно со скобочной фразовой структурой текста процедуры. Объект можно создавать динамически, и его время жизни может быть больше или меньше времени выполнения фразы, внутри которой он создан. Например, внешний объект может существовать и после того, как завершилось выполнение программы, которая его создала. Иными словами, стратегия распределения ресурсов для объектов имеет динамический характер.

Базовые понятия и их взаимосвязь имеют определенную интерпретацию и для внешних объектов. В автокоде существует три основных типа внешних объектов: файл, архивный справочник, контейнер (семейство пакетов дисков или бобин магнитных лент).

Внешние объекты можно создавать и ликвидировать в процессе выполнения программы; обозначение, описанное в программе, или имя можно связать с внешним объектом. Т. е. программа, наряду с оперативными объектами, может обычными средствами взаимодействовать и с внешними объектами. Понятия, изображенные на рис. 1.2, трактуются для пространства внешних объектов следующим образом:

— роль процедуры играет программа. Текст программы — это способ обозначения ее алгоритма. Для программы, как и для процедуры, определена операция вызова;

— контекстом обозначений для программы является совокупность доступных ей внешних обозначений. В тексте программы такие обозначения используются для идентификации внешних объектов;

— контекст программы (так называемый внешний контекст) складывается из контекста внешних обозначений и множества доступных внешних объектов;

— две различные программы могут, вообще говоря, иметь различный внешний контекст;

— если во внешнем контексте одной программы находится другая программа, то первая может вызывать вторую как обычную процедуру. Для этих целей используется традиционная конструкция «вызов процедуры».

4.2. Динамика типов. Прежде всего уточним смысл терминологии, введенной в § 2.

Тип. Набор встроенных типов определяется номенклатурой операций, реализованных аппаратными средствами, и прими-

тивов, реализованных операционной системой. Тип стандартным образом кодируется в информационном содержимом объекта и аппаратно распознается в процессе обработки объекта, т. е. в данном случае существует непосредственная связь между объектом и его типом.

Простые и составные объекты. Совокупность типов простых объектов включает в себя стандартные типы распространенных ЯВУ. В их число входят целые и вещественные числа, логические и литерные значения, процедуры, указатели составных объектов.

Среди составных объектов различаются оперативные (располагающиеся в оперативной памяти) и внешние. Типичный представитель оперативных объектов — массив, а типичный представитель внешних объектов — файл. С составным объектом всегда связан указатель, описывающий данный составной объект.

Например, с массивом связан указатель массива (см. § 2). Указатель идентифицирует соответствующий составной объект и является его «представителем» в операциях, т. е. в операции обработки составных объектов в качестве операнда передается не сам объект, а указатель.

Наличие указателей означает, что в языке допускается обработка адресной информации. Введение адресной информации — это всегда серьезный шаг в конструировании языка, поскольку он требует решения связанных с ним вопросов защиты. Дело в том, что основным источником ошибок, приводящих к нарушению защиты, являются некорректные действия с адресной информацией.

В автокоде средства обработки указателей устроены так, что их использование не может повлечь за собой нарушение защиты. Причем, как будет показано ниже, это свойство языка основывается на механизме, отличном от статического контроля типов. Существенно подчеркнуть еще и то обстоятельство, что в автокоде введены не только указатели по оперативной памяти, но и указатели по внешней памяти (указатели внешних составных объектов). Это увеличивает гибкость и возможности языка, но предъявляет дополнительные требования к системе в части реализации ссылок и защиты в адресном пространстве внешних носителей.

Операции. В МВК Эльбрус аппаратно реализованы универсальные операции. В процессе выполнения операции осуществляется анализ типов и выбор конкретного алгоритма в зависимости от типов операндов. Одновременно производится контроль правильности типов.

Номенклатура операций включает в себя основные операции ЯВУ: арифметические, логические, присваивание, индексация, вызов процедуры и пр. Также представлены операции над литерами и строками, средства обработки упакованной информации. Таким образом, можно сказать, что, с точки зрения семантики, язык МВК Эльбрус — это не традиционный бестиповый язык низкого уровня, а язык высокого уровня класса Д.

Имена. Имя не связано с типом именуемого объекта и может именовать объект любого типа (последний всегда является простым, поскольку доступ к составным осуществляется косвенным образом, через посредство указателя). В соответствии с этим при описании переменной тип ее возможных значений не указывается. Например, в результате выполнения описания

ф64 *x*;

создается имя, с которым связывается обозначение *x*. Ключевое слово, предшествующее идентификатору, задает размер памяти, отводимой при создании имени. Размер этот называется форматом. Наиболее распространенный формат ф64 соответствует 72-разрядному слову памяти МВК Эльбрус (64 информационных разряда + 8 служебных; тип объекта или объектов, помещенных в слово, кодируется в служебных разрядах).

Массивы. Массив создается динамически с помощью генератора массива. Так как тип именуемых объектов в языке не фиксируется, его не надо задавать при генерации массива имен. В общем случае в массиве может находиться разнотипная информация. Приведем пример создания массива:

x := лок вект [100] ф64 ;

Здесь создается массив из 100 элементов формата ф64; указатель массива присваивается переменной *x*. Обращение к элементам массива изображается обычным образом:

x[*i*] := 3.14

Указатель массива содержит информацию о формате элементов и длине массива. В соответствии с этим аппаратно реализованная операция индексации в зависимости от формата вычисляет смещение элемента относительно начала массива и осуществляет контроль границ.

Пример. Чтобы дать начальное представление об изобразительных средствах языка и проиллюстрировать простейшее применение динамики типов, запрограммируем на автокоде пример 1.6 § 3.

Пример 1.

```
проц (ввфайл , выфайл) % эта программа является процедурой
                        % с двумя параметрами:
                        % ввфайл — файл ввода,
                        % выфайл — файл вывода
начало
  конст a = лок вект [100] ф64 ; % «описание» массива
  процедура сумма = функция (в)
    начало
      ф64 с := 0;
      для k от 0 до длина в — 1
      цикл
        с := с + в [k]
      повторить ;
      с % значение процедуры — функции
    конец; % процедуры
  читф (ввфайл , а) ;          % ввод в массив а
  запф (выфайл , сумма (а)) % вывод результата
конец
```

Операторы *читф* и *запф* выполняют форматный обмен, управляемый данными. Целые и вещественные числа, находящиеся во входном файле, помещаются в массив *а* без преобразования типа. Оператор *запф* анализирует тип результата и выводит его в виде целого или вещественного числа.

Операция сложения, находящаяся внутри цикла, универсальная. Она выполняет именно те действия, которые в приводимой ранее программе алгола-68 представлены вложенными сопоставляющими предложениями. Если все вводимые числа — целые, то результат суммирования будет целочисленным.

Тип значения процедуры-функции не фиксируется в описании. Он будет таким, какой получится в результате выполнения вычислений, производимых в теле процедуры.

Таким образом, изображенная программа достаточно лаконичными средствами решает ту простую задачу, которая ставилась в п. 3.1: во-первых, в массив *а* вводится то, что поступает с входного файла, без неявного преобразования типа, и во-вторых, в случае, если все вводимые числа — целые, выводимый результат также будет иметь тип целое.

Ниже будет рассмотрен пример вызова этой программы.

4.3. Внешние объекты. С точки зрения практической деятельности программиста в каждый момент времени существует набор доступных ему взаимосвязанных внешних объектов, в число которых входят архивные справочники, файлы, программы. Некоторые программы находятся в процессе выполнения. При

этом они взаимодействуют с другими внешними объектами. Обособление внешних объектов приводит к концептуальному разрыву между этой реальной картиной и тем, как она отображается в тексте программ пользователя. Выше были рассмотрены конкретные примеры того, как отсутствие логического единства приводит к усложнению программирования. Здесь мы обсудим трактовку внешних объектов в автокоде.

Чтобы программа могла оперировать с пространством внешних объектов, надо ввести в язык типы этих объектов, операции над ними и способ обозначения.

Файлы: Очевидно, что в первую очередь вводятся объекты типа файл и соответствующие операции обмена. Забегая вперед, отметим, что все составные объекты создаются динамически с помощью генераторов. Это же касается и объектов типа файл.

Пример 2. Создание файла.

```
начало
   ф64 x;
   процедура p = проц (y)
   начало
      ...
      вап (y , ... параметры ...) ; % оператор обмена
      ...
   конец ;
   ...
   x := файл (... атрибуты ...) ; % создать файл
   p (x) % передать файл, как параметр
   ...
конец
```

Как уже отмечалось, представителем любого составного объекта является указатель соответствующего типа. В приведенном примере такой указатель присваивается переменной x и затем передается в качестве параметра процедуре p. Во всех последующих примерах, связанных с внешними объектами, подразумевается такой же механизм идентификации объектов. Для определенности заметим, что указатель занимает одно слово памяти.

Файл можно обозначить таким же образом, как это делается для чисел, т. е. изобразив его содержимое. Например, в результате присваивания:

```
x := тфайл
   *!!*
   содержимое
   файла
   !!
```

значением переменной x становится указатель текстового файла, содержимое которого заключено в ограничители *!!* и !!.

Тип файла. Существенным атрибутом файла является атрибут «тип файла», определяющий логическую структуру содержащейся в нем информации. В автокод встроен ряд стандартных типов. В этом разделе целесообразно упомянуть две разновидности: текстовый файл и файл объектного кода.

Текстовый файл состоит из литерных строк переменной длины. Одно из наиболее распространенных применений текстового файла — это представление текста программ и входных данных. Выше приведен пример того, как в автокоде можно изобразить такой файл.

В результате трансляции файла текста программы образуется файл объектного кода. Такой файл можно а) читать и модифицировать и б) «выполнять». Чтобы выполнить файл, к нему надо применить операцию «вызов процедуры».

Файл объектного кода трактуется как объект-оболочка, внутри которого находится объект-содержимое. Последний называется программой. Объект-содержимое (программа) отличается от файла объектного кода тем, что его можно только выполнять, но нельзя читать и модифицировать.

Существует возможность создать указатель для доступа к объекту-оболочке. Такой указатель разрешает также обработку объекта-содержимого. Но можно создать указатель для доступа только к программе. В операциях обмена подобный указатель запрещен. Это позволяет ликвидировать возможность несанкционированного изменения кода программ. Один и тот же код может в одном контексте фигурировать как файл, а в другом — как программа. Тогда процедуры и программы, выполняющиеся во втором контексте, могут применять только операцию «вызов процедуры».

Обмен. Разработка средств организации обмена преследовала двоякую цель. С одной стороны, нужно было обеспечить достаточно высокий уровень программирования обменов, т. е. ликвидировать необходимость явным образом распределять внешние устройства, распределять память на внешних устройствах, формировать заявку на конкретный обмен и т. д. С другой стороны, необходимо, чтобы эти же средства были достаточно близки к уровню внешних устройств. Это, во-первых, дает возможность повысить эффективность и простоту реализации, во-вторых, позволяет с помощью этих же средств программировать обмены операционной системы и, в-третьих, обеспечивает основу для построения других систем файлов, в частности тех, которые описаны в языках кобол, PL/1 и алгол-68.

В соответствии с изложенными целями в автокод введены следующие разновидности обмена:

— буферизованный двоичный обмен. Этот вид обмена можно трактовать таким образом. Существует «окно», через которое можно просматривать (читать/модифицировать) блоки файла или добавлять в файл новые блоки. Таким окном является буфер, выделяемый системой неявным образом. С помощью соответствующих операций окно можно перемещать по файлу. Существует возможность иметь одновременно несколько подобных окон для обработки различных участков одного и того же файла. Предварительная подкачка блоков в оперативную память и запись модифицированных и новых блоков в файл осуществляется системой неявно и параллельно с выполнением программы. Семантика операций буферизованного обмена практически полностью инвариантна относительно типов устройств;

— буферизованный обмен с редактированием. Этот вид обмена по смыслу близок к традиционному форматному обмену;

— непосредственный двоичный обмен. В этом виде обмена происходит передача информации между программными массивами и внешним носителем без какой-либо промежуточной буферизации и редактирования.

Справочники. Для расширения понятия «контекст» на пространство внешних объектов необходимо (1) обобщить понятие «имя» и (2) определить средство построения внешнего контекста. С этой целью вводится объект типа справочник. По своим логическим свойствам справочник напоминает массив. Он состоит из набора имен, каждое из которых может именовать внешний объект. Пара (имя, именуемый объект*)) образует элемент справочника. С помощью операции *запспр*, похожей по смыслу на присваивание, можно изменять текущее значение элемента справочника (пример см. ниже). Элемент справочника идентифицируется с помощью алфавитно-цифрового обозначения.

Пример 3. Создание справочника.

```
начало
  ф64 спр ;
  процедура р = проц (у) начало ... конец ;
  ...
  спр := генспр (    ) ; % создать справочник
  создэлспр (спр , "ф1") ; % создать элемент справочника,
                           % обозначив его "ф1"
```

*) В данном случае именуемый объект — это указатель (ссылка) на файл.

вапспр (*спр*//*ф1* , *файл* (...)) ; % «присвоить» элементу
% справочника новый файл
р (*спр*//*ф1*) ;
...
конец

Здесь конструкция *спр*//*ф1* используется по аналогии с переменной с индексами. Она служит для обозначения имени элемента справочника. В этом примере два оператора *создэлспр* и *вапспр* можно заменить на один, производящий одновременно создание элемента и его инициализацию:

создэлспр (спр, "*ф1*", *файл* (...))

В дальнейшем для определенности будем говорить, что элемент справочника содержит «ссылку» на именуемый объект.

Программа, контекст программы. С точки зрения использования объект-программа должен быть эквивалентен процедуре. Конкретно это выражается в том, что (1) программа вызывается теми же средствами, что и процедура, и (2) для программы определено понятие контекст. Ниже будет показано, что в системе Эльбрус эквивалентность программ и процедур является следствием общей однородности понятий языков высокого уровня и понятий системы. Здесь остановимся на определении контекста программы.

Все внешние объекты, используемые программой, можно по аналогии с процедурой разбить на три категории: объекты, создаваемые в процессе выполнения, объекты, переданные в качестве параметров, и объекты, составляющие статическое окружение. Последние образуют внешний контекст программы. Это может быть весь контекст пользователя, разработавшего программу, или ограниченная часть контекста. Связывание программы с ее контекстом осуществляется либо автоматически, либо явно путем применения специальной операции.

Рассмотрим, как возникает внешний контекст программы. Существует специальное средство языка, позволяющее связать программу с каким-либо справочником. Справочник, связанный с программой, называется корневым справочником внешнего контекста данной программы. Обозначения элементов этого справочника составляют контекст внешних обозначений для программы. В тексте программы корневой справочник обозначается стандартным идентификатором *свойвк*.

Пример 4.

начало % создать элемент в корневом справочнике
...
создэлспр (*свойвк* , ''*a*'' , *файл* (...)) ;

```
р (свойвк //а)
...
конец
```

Элемент корневого справочника можно обозначать короче: не *свойвк//а*, а просто *//а*.

Совокупность контекста внешних обозначений и внешних объектов, достижимых через посредство элементов корневого справочника, называется внешним контекстом программы. Поскольку значением элемента справочника может в свою очередь быть объект типа справочник, внешний контекст может представлять собой многоуровневую иерархическую систему справочников. Элементы этой системы идентифицируются многослоговыми обозначениями, описывающими путь от корневого справочника, например:

//спра//спрв//х.

Как возникает изначальный внешний контекст пользователя? При регистрации пользователя в системе неявно создается справочник, называемый корневым справочником пользователя. Любая программа пакетного задания данного пользователя перед выполнением автоматически связывается с его корневым справочником. Следовательно, она может создавать в нем новые элементы, присваивать им новые внешние объекты и т. д. В частности, приведенный выше пример можно рассматривать, как программу задания. Пользуясь традиционной терминологией, можно сказать, что эта программа создает в архиве файл с архивным именем *а*.

Вновь созданную программу можно связать с корневым справочником пользователя. Тогда ей будет доступен весь его архив. Но можно связать программу со справочником, через посредство которого доступна только часть архива. Таким образом, потенциально существует средство, с помощью которого можно управлять объемом части архива, доступной программе пользователя. Связь запоминается в объекте-программе и используется при всех последующих вызовах. Следовательно, отпадает необходимость каждый раз при вызове программы указывать ее статическое окружение, как это, например, имеет место в ОС 360. С другой стороны, это обеспечивает межпрограммную защиту: в контексте вызывающей программы может быть известно только обозначение вызываемой программы, но не ее внешний контекст. Объекты, не входящие в статическое окружение программы, могут быть переданы ей в качестве фактических параметров.

Пример наращивания контекста. Опишем задание, выполняющее такую последовательность действий: трансляция текста программы суммирования, приведенной в п. 4.2, и запись файла кода в архив. Будем предполагать, что: а) внешнее обозначение программы автокод-транслятора — *//авт*; б) параметром транслятора является файл текста транслируемой программы; в) транслятор описан как процедура-функция, значением которой является результирующий файл объектного кода.

Пример 5.

```
начало
создвлспр (свойвк , "провсум" ,
      //авт (тфайл
          *!!*
          текст
          программы
          !!
             ))
конец
```

После выполнения этой программы результирующий файл кода можно идентифицировать обозначением *//провсум*. Последующий вызов программы *//провсум* можно изобразить таким образом:

Пример 6.

```
начало
  конст ввод = позп (тфайл
                     *!!*
                     десять
                     чисел
                     !!) ,
     вывод = позп (ацпу);
  //провсум (ввод , вывод)
конец
```

Здесь учтено, что в форматном обмене требуется, чтобы файл был предварительно открыт для буферизованного обмена. Это выполняется с помощью стандартной функции *позп*. Во втором описании открывается вновь созданный файл на алфавитно-цифровом печатающем устройстве.

Неявные преобразования. Средства динамического анализа типов обрабатываемых объектов позволяют в определенных случаях выполнять неявные и достаточно очевидные преобразования. Например, если в операцию «вызов процедуры»

подана не программа, а текстовой файл, то система неявно вызывает транслятор и передает ему файл на трансляцию. После получения результирующей программы операция «вызов процедуры» повторяется, в результате чего и происходит вызов нужной программы. По атрибутам файла можно установить, на каком языке программирования написан текст программы. Это позволяет системе выбрать нужный транслятор.

Указанное свойство, в частности, используется для сокращения записи программ заданий: можно не описывать трансляцию проблемной программы, а просто использовать в конструкции «вызов» ее текстовый файл. Например, задание может выглядеть так:

```
(тфайл (имяяз : ''ав п'')
*!!*
... текст программы ...
!!
(... параметры ...))
```

Вместо слов **начало** и **конец** можно использовать скобки.

Существует ряд других неявных преобразований внешних объектов (подробно см. описание языка).

4.4. **Единицы вычислений.** Действия, производимые над объектами, будем называть вычислениями. Обычно в качестве элементов вычислений рассматриваются процесс, выполнение процедуры и вычисление выражения. В МВК Эльбрус процессу соответствует аппаратно реализованный стек. Переключение процессора с одного процесса на другой осуществляется специальной операцией.

В стеке располагаются локальные данные активаций процедур, вызываемых в рамках данного процесса. Процедурный механизм, включая смену контекста, передачу параметров и отведение локальной памяти в стеке, реализован аппаратными средствами. Верхняя часть стека выполнена на регистрах; здесь по известному магазинному принципу происходит вычисление выражений. Соответственно, код выражений представлен в виде постфиксной польской записи. Таким способом в аппаратуре отображены понятия, связанные с динамикой выполнения алгоритма. Аналогичные элементы вычислений существуют и в автокоде:

— вызов процедуры. Эта операция активирует выполнение процедуры в рамках текущего процесса;

— запуск процесса. В результате создается новый процесс, начинающийся с выполнения заданной процедуры.

Прежде чем перейти к рассмотрению еще одного элемента вычислений — задачи, надо пояснить ряд деталей, связанных с реализацией в системе Эльбрус объектов-программ.

Программа. В распространенных языках нет достаточно четкой формулировки понятия программы. В языках, ориентированных на машины фирмы IBM, существует термин «выполняемая программа», т. е. здесь под программой понимается совокупность модулей, вызывающих друг друга в процессе выполнения. Такая трактовка не является вполне определенной. Она, в частности, может подразумевать динамическое разрешение ссылок на оперативные объекты (общие области, общие переменные). Последнее, с одной стороны, приводит к потере эффективности, и, с другой стороны, означает использование метода динамической известности имен, что может вызвать случайное совмещение объектов, имеющих разное назначение. Поэтому реально программист должен в программе задания, т. е. статически, указать редактору связей предполагаемую совокупность взаимодействующих модулей.

Языки алгол и его последователи придерживаются противоположного подхода: здесь программа представляет собой замкнутый блок без внешних связей. Это означает, что средствами языка невозможно, например, описать библиотеку параметризованных программ *), а также вызов библиотечных подпрограмм. С этой целью в конкретных реализациях в язык вводятся дополнительные средства.

Таким образом, языки программирования лишь номинально определяют форму программы. Фактическое определение дает конкретная система в зависимости от ее возможностей. Причина этого явления состоит в том, что внешний уровень программы находится на стыке проблемного алгоритма и системы. Непрерывность этого перехода зависит от решения ряда ключевых проблем, определяющих архитектуру системы. Сюда относятся механизмы адресации, распределения памяти, именования внешних объектов, организации межпрограммных и межмодульных связей. Если эти вопросы однородно решаются на уровне языка и на уровне системы, то программа эквивалентна процедуре. В противном случае нужна особая трактовка.

Например, способ, которым в традиционных системах программирования реализован механизм вызова процедуры, отли-

*) В паскале можно в заголовке программы описать формальные параметры — файлы. Но вопросы вызова одной программы из другой и передачи параметров программе оставлены за пределами определения языка.

чается от способа вызова программы в операционной системе. В первом случае — это переключение контекста, загрузка фактических параметров в известную область и переход; во втором случае — статическое планирование оверлейной структуры, загрузка кода, особая организация механизма связи с внешними объектами.

Аппаратная реализация процедурного механизма позволяет сравнительно простыми средствами привести вызов программы к вызову процедуры. При вызове программы в системе Эльбрус автоматически выполняются два дополнительных действия: (1) в произвольное место оперативной памяти помещается справочник процедур программы, где для каждой процедуры указано ее относительное местоположение в файле объектного кода программы; (2) осуществляется переключение внешнего контекста. В остальном аппаратно производятся те же операции, что и при вызове процедуры.

Попутно отметим, что в системе нет фазы загрузки кода программы. Если в момент вызова процедуры оказывается, что ее кода нет в памяти, то код выбирается в произвольную свободную область. При этом его коррекция не производится, так как код МВК Эльбрус — перемещаемый, он не содержит абсолютных адресов.

Параметризация программы осуществляется с помощью обычного процедурного механизма передачи параметров. В частности, если фактическим параметром программы является внешний объект, то передается указатель этого объекта. Примеры вызова программ приведены выше (см. примеры 5, 6). Механизм вызова программ и процедур одинаковый как на системном уровне, так и на уровне пользователя.

Учитывая эквивалентность программ и процедур, а также то, что внешние объекты допустимы в языке, можно в простейшем случае дать следующее определение: автокод-программа — это процедура, являющаяся внешним объектом. Тем самым подчеркивается, что программа обладает и свойствами процедуры, т. е. к ней применимы операции вызова процедуры и запуска процесса, и свойствами, которыми обладает внешний объект.

Задача. Рассмотрим еще один элемент вычислений, который находится на первом уровне динамической иерархии элементов вычислений; остальные являются динамически подчиненными. Особенность этого элемента состоит в том, что он однозначно идентифицирует пользователя, от имени которого производятся вычисления. Эта связь используется при начислении расхода ресурсов.

Несмотря на то, что такой элемент всегда присутствует в вычислениях, в языках высокого уровня он не имеет соответствующего отображения; для описания его алгоритма используется специальный язык, например, язык управления заданиями. Причина этого аналогична причине обособления внешних объектов: на первом уровне вычислений используются понятия, свойственные конкретной операционной системе. Текст программы первого уровня воспринимается системой особым образом: здесь применяется специальная интерпретационная техника выполнения.

В системе Эльбрус элемент первого уровня называется задачей. Каждая задача выполняется в собственной математической памяти. В МВК понятию «задача» соответствует аппаратная реализация механизма математической памяти.

Чтобы создать задачу, надо к программе применить операцию «запуск задачи». Для этого используется соответствующая конструкция автокода. В результате создается новая математическая память, в которой начинает выполняться заданная программа как начальный процесс. Поскольку программа связана с корневым справочником доступного ей пространства внешних объектов, одновременно происходит переключение внешнего контекста. Выполняемая программа может, в свою очередь, вызывать процедуры, создавать процессы и задачи.

Изложенное, в частности, означает, что в системе отсутствует необходимость иметь специализированный язык управления заданиями. Роль программы задания может выполнять автокод-программа, в которой средствами языка описано взаимодействие с нужными архивами, файлами и программами. При поступлении очередного задания операционная система вызывает автокод-транслятор для трансляции текста программы задания, затем связывает результирующую программу с корневым справочником внешнего контекста пользователя, составившего задание, и применяет к ней операцию «запуск задачи». Отметим попутно, что выполняя все эти действия, операционная система не выходит за рамки понятий языка.

4.5. Динамическая генерация объектов. В автокоде средством создания объектов является конструкция «генератор». Генератор — это не описание, а частный случай выражения. С его помощью составной объект любого типа может быть создан в произвольный момент выполнения программы, т. е. в языке реализована динамическая концепция генерации объектов. Отметим, что такой подход хорошо согласуется с динамикой типов; наличие статических описаний объектов означало бы фиксацию типа переменных.

При введении в язык динамических генераторов надо решить вопрос о том, как определяется время жизни объекта. Наиболее естественным является принцип «удерживания» (retention [32]) объекта до тех пор, пока существуют ссылки на него. Надо учесть, однако, что реализация этого принципа требует дополнительных накладных расходов *).

Из соображений эффективности целесообразно в этом вопросе отдельно рассматривать оперативные объекты и отдельно — внешние. Обращение к оперативным объектам происходит значительно чаще, чем к внешним, и время доступа к ним значительно меньше. Поэтому дополнительные накладные расходы здесь неприемлемы. В случае внешних объектов ситуация иная, и здесь можно применять особые методы.

Оперативные объекты. Известно, что процедуры из соображений модульности программируются так, чтобы большинство создаваемых ими оперативных объектов существовало (и использовалось) не дольше, чем они выполняются. По этой причине в автокоде оперативные объекты разделяются на локальные и глобальные. Локальные объекты уничтожаются при выходе из той процедуры, где они были созданы; глобальные уничтожаются по окончании задачи. Дополнительно существуют средства явного уничтожения объектов.

Динамическая генерация объектов вытекает из общего принципа динамического заказа ресурсов в системе Эльбрус. Это, в частности, касается и ресурсов оперативной памяти. Динамика памяти поддержана целым рядом аппаратно-программных механизмов. Кратко перечислим их.

На оперативную память наложена аппаратно реализованная математическая память. При генерации массива отводится математическая память нужного размера. Участки оперативной памяти выделяются при первом обращении к соответствующим элементам массива. Поиск участка нужного размера осуществляется с помощью специальных аппаратных операций поиска по спискам и таблицам. Наконец, для обеспечения механизма листания используется специально разработанная быстрая вторичная память (барабаны системы Эльбрус).

---

*) В этих случаях применяется, например, метод «сборки мусора» или метод подсчета числа ссылок. Первый метод предполагает, что время от времени запускается системный алгоритм, выполняющий поиск и уничтожение таких объектов, на которые не осталось ссылок. Во втором методе объект содержит счетчик числа ссылок на него. Объект уничтожается при обнулении счетчика.

Стратегия локализации объектов также отображена в системе: при выходе из процедуры, имеющей локальные составные объекты, возникает особая ситуация, при которой аппаратно вызывается системная процедура, осуществляющая освобождение памяти.

Внешние объекты. При решении вопроса о времени жизни внешнего объекта надо учесть его качественные и количественные отличия от оперативного объекта. Внешний объект является более стабильным образованием, чем оперативный, и занимает, как правило, большой объем памяти. Если его время жизни назначается статически, то описание файла должно предсказывать последовательность обработки.

В качестве примера достаточно рассмотреть параметр *disp* из *dd*-предложения языка JCL/360. Этот параметр указывает, является ли файл новым или уже существующим, сохранить ли файл, уничтожить, передать в другой пункт задания или уничтожить по завершении всего задания. Такая детализация затрудняет обработку файлов.

Достоинство принципа удерживания состоит в том, что время жизни объекта определяется неявно, исходя из динамики его использования. С другой стороны, в случае внешних объектов накладные расходы на реализацию удерживания малы по сравнению с общими расходами. Все эти соображения говорят о том, что для внешних объектов целесообразно реализовать именно этот принцип.

В системе Эльбрус время жизни внешнего объекта определяется, исходя из подсчета общего числа объектов, ссылающихся на данный. При обнулении числа ссылок объект уничтожается. Простые примеры удерживания объектов уже были приведены ранее. Действительно, файл, создаваемый в примере 2, будет уничтожен по окончании выполнения программы. Причина в том, что ни один из внешних объектов, продолжающих существовать после завершения выполнения, не ссылается на этот файл.

Иные обстоятельства в примере 4. Здесь создается файл, ссылка на который помещается в элемент корневого справочника. Этот справочник (а следовательно и ссылка) продолжает существовать после окончания программы. Поэтому файл будет сохранен.

В системе Эльбрус динамическая генерация внешних объектов поддерживается динамическим распределением внешних устройств и памяти на носителях. При создании файла для него отводится минимально необходимое пространство. По мере необходимости это пространство увеличивается. С этой целью

на дисках и барабанах используется листовая структура памяти; размер листа задается при генерации файла или выбирается по умолчанию.

4.6. Контекстная защита. Теперь мы накопили достаточное число сведений для того, чтобы дать итоговый обзор свойств языка, обеспечивающих защиту объектов. Этот обзор одновременно позволит сконцентрировать внимание на двух механизмах, введенных в системе Эльбрус. Первый из них — это аппаратно реализованная контекстная защита оперативных объектов, и второй — многоуровневый архив с контекстной защитой.

Контекст программы пользователя строится из стандартного контекста и внешнего контекста:

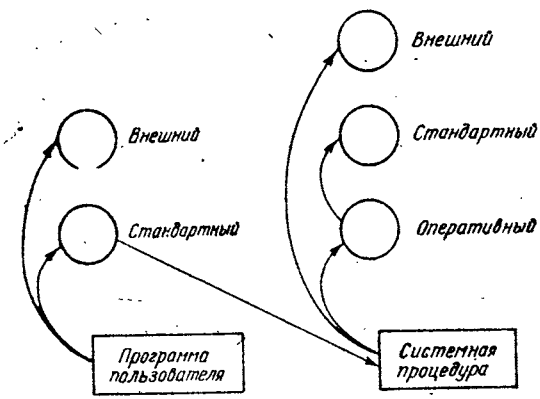


Рис. 1.3. Контекст программы.

В стандартном контексте находятся аппаратно реализованные операции и процедуры операционной системы, доступные пользователю. На рис. 1.3 изображена ссылка из стандартного контекста на процедуру. Ее можно понимать как связь между обозначением (в стандартном контексте) и обозначаемым объектом (системной процедурой).

Контекст системной процедуры строится из оперативного контекста, где находятся оперативные объекты, общие для процедур операционной системы (массивы, имена переменных и пр.), стандартного и внешнего. В стандартном контексте некоторых системных процедур, кроме обычных операций, находится ряд так называемых привилегированных операций. И те и другие реализованы в машине, но аппаратура устроена таким образом, что в стандартном контексте пользователя привилегированные операции недоступны.

Разберем пример, иллюстрирующий одно из типичных применений контекстной защиты. Пусть в стандартном контексте программы П пользователя находится операция индексации и системная процедура *генмассив*, осуществляющая динамическую генерацию массива. Перечислим обстоятельства, объясняющие, почему с помощью операции идексации нельзя нарушить защиту:

1) процедуры программы имеют доступ только к тем массивам, которые ими созданы через посредничество процедуры *генмассив*. Других массивов в контекстах процедур программы нет. При выполнении индексации проверяется, что индекс не выходит за границы, заданные в момент создания массива;

2) указатель массива содержит информацию о реальном местоположении массива в памяти и о числе элементов. Если уметь произвольным образом менять эту информацию, то можно, например, попытаться наложить этот указатель на какой-либо другой массив, находящийся в памяти, но недоступный программе П. В результате подобных действий второй массив станет доступен для П. Оказывается, что это невозможно сделать, так как в контексте П нет операций, предназначенных для изменения информации в указателе. Есть лишь одна операция, связанная с обработкой массивов,— индексация. Но она обеспечивает доступ к элементам массива, а не к информации в указателе. Никакая другая операция из числа доступных П для указанных целей также не подходит: операции контролируют типы операндов. В частности, попытка применить к указателю какую-либо операцию, не связанную с этим типом, приведет к диагностируемой ошибке;

3) можно попытаться осуществить «подделку», т. е. сначала сформировать объект, похожий по структуре на указатель, а затем преобразовать его к типу указатель. Это также позволило бы получить доступ к любому участку памяти. Но в контексте П нет такой операции преобразования. Если же не преобразовывать в указатель, то ошибка будет обнаружена операцией индексации, которая, как и все операции, контролирует тип операндов.

В контексте процедуры *генмассив*, кроме операции индексации, есть еще и ряд других операций, связанных с массивами. В частности, там есть и операция изменения информации в указателе. Тот факт, что *генмассив* имеет доступ ко всем существующим массивам, вполне объясним. Эта процедура обеспечивает распределение памяти и, следовательно, обладает особыми привилегиями.

Можно привести аналогичные примеры, касающиеся обработки внешних объектов. В стандартном контексте пользователя известны лишь процедуры создания/ликвидации объектов и «безопасные» операции взаимодействия. Но операции, позволяющие изменять служебную информацию, содержащуюся в объекте, например, информацию о реальном местоположении файла на внешнем носителе, недоступны. Как и в случае массивов, программа пользователя не может, минуя систему файлов, сформировать указатель файла и получить тем самым доступ в произвольной области внешней памяти. Иными словами, система файлов обрабатывает только те указатели, которые она же и создала.

Подытоживая, еще раз отметим, что защиту в автокоде (и в системе Эльбрус в целом) обеспечивают два механизма: а) контекстная известность объектов (в частности, операций) и б) контроль типов. Важно подчеркнуть, что оба механизма реализованы аппаратными средствами, а не интерпретационно, что существенно с точки зрения эффективности.

Попутно отметим еще одно обстоятельство. Предположим, что защита в системе основывается на статическом контроле типов во время трансляции. Тем самым неявно предполагается, что используются только отлаженные трансляторы. Ошибка в работе транслятора может привести к тому, что будет создан такой код, который при выполнении нарушит функционирование системы в целом. Иными словами, защита в подобной системе основывается на отлаженных трансляторах (как, например, в системе Burroughs [27]). Если же защита реализована аппаратными средствами, то любой транслятор можно рассматривать как обычную программу пользователя. Созданный им код может содержать ошибки. Выполнение этого кода в худшем случае приведет к аварийному завершению работы данной программы, но не нарушит функционирование системы в целом. Это одно из тех важных свойств, которые обеспечивают живучесть системы Эльбрус.

В заключение рассмотрим использование контекстной защиты в пространстве внешних объектов. На рис. 1.4 изображен пример структуры внешнего контекста пользователя. Такую конфигурацию можно создать, используя упоминавшиеся выше примитивы автокода, связанные с обработкой внешних объектов. Имена элементов корневого справочника КА пользователя А именуют три программы: a_1, a_2 и a_3; кроме того, один из элементов КА содержит ссылку на справочник глобальных объектов, доступных всем пользователям. В их число входят трансляторы, пакеты стандартных функций и пр. Эта связь создает-

ся системой при генерации корневого справочника пользователя.

Корневым справочником для a_1 является справочник КА. Программы a_2 и a_3 имеют собственные корневые справочники КА2 и КА3, причем в каждом из них есть ссылка на КА. Далее, элементы КА2 ссылаются на внешние объекты, недоступные через КА3. Их назовем приватными для программы

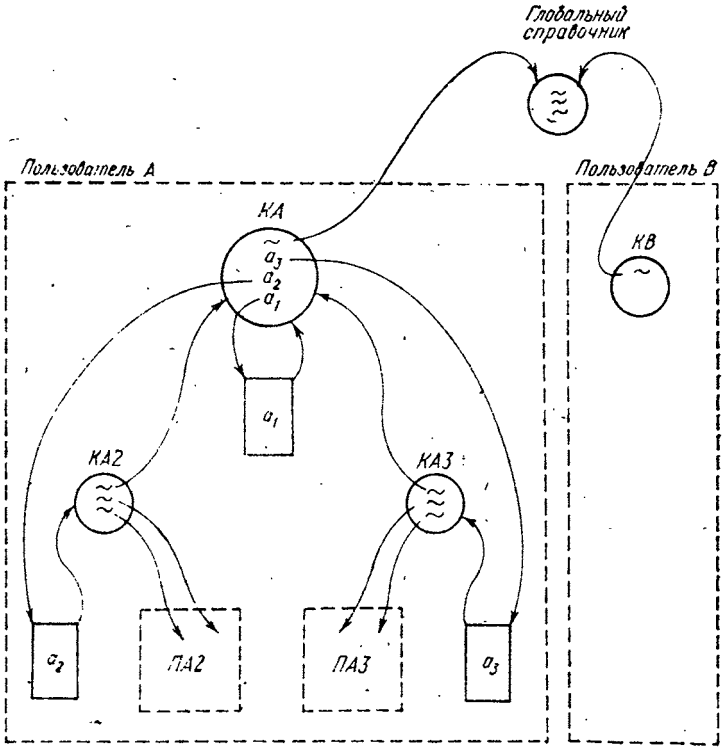


Рис. 1.4. Контекстная защита в пространстве внешних объектов.

a_2 (ПА2). Аналогично элементы КА3 ссылаются на объекты, недоступные через КА2.

Таким образом, программа a_1 может использовать только программы a_2 и a_3, но объекты ПА2 и ПА3 ей недоступны. Программа a_2 может использовать свои приватные объекты ПА2 (возможно,— опять программы) и программы a_1 и a_3, но ей недоступны приватные объекты программы a_3. Аналогично

программа a_3 может использовать объекты ПА3, a_1 и a_2 (рекурсивный вызов программ, разумеется, также возможен), но не объекты ПА2. Объекты, на которые ссылаются элементы глобального справочника, доступны всем трем программам.

Отметим еще одно обстоятельство. Пусть в КА1 и КА2 есть элементы с одинаковым обозначением, и пусть это обозначение используется в тексте a_2 и в тексте a_3. При вызове одной программы из другой путаницы не произойдет: каждая из них будет обращаться к своему приватному объекту. Это следует из принципа статической контекстной ассоциации обозначений. В случае обозначений, вводимых в программе с помощью описаний, статическая ассоциация осуществляется, исходя из текста программы. Когда рассматривается пространство внешних объектов, то статика состоит в том, что контекстные связи программы фиксируются в момент создания ее, как объекта, и не изменяются при всех последующих вызовах.

В правой части рисунка условно обозначено пространство внешних объектов другого пользователя. Взаимосвязь объектов в этом пространстве может быть такая же, как у пользователя А или какая-либо другая. Важно отметить, что общими для них являются только глобальные объекты. Пользователь А не имеет доступа к объектам пользователя В и наоборот. Это, однако, не мешает создавать собственные подсистемы, считая, что пользователь — это целая группа лиц. При этом надо соответствующим образом структурировать архив такого коллективного пользователя, чтобы обеспечить персональную защиту.

4.7. Полная рекурсивность языка. Автокод определен полностью рекурсивно. Операнды любой конструкции могут быть представлены выражениями произвольной сложности. Например, присваивание поставляет значение и поэтому может являться операндом в некоторой другой операции: левый операнд индексации может быть представлен не только идентификатором, но и выражением, поставляющим указатель массива, получаемый в результате целой серии вычислений; левая часть присваивания может быть представлена условным и выбирающим предложением и т. д., т. е. язык позволяет адекватно отобразить рекурсивную фразовую структуру алгоритма.

Это свойство является одним из примеров отображения высокого семантического уровня системы команд и архитектуры МВК. Во-первых, динамика типов и динамическая идентификация операций ликвидируют трудности, связанные с балансировкой типов (см. п. 8.4). Во-вторых, аппаратная реализация базовых понятий ЯВУ обусловливает простоту трансляции ре-

курсивных конструкций и эффективность результирующего кода.

Например, понятию выражение в машине соответствуют: (1) код в форме постфиксной польской записи, (2) стек, в котором вычисление выражений происходит по известному магазинному принципу. Верхняя часть стека выполнена на быстрых регистрах. Распределение регистров осуществляется динамически, в процессе вычисления выражения, а не статически, во время трансляции. Таким образом, задача транслятора состоит только в переводе выражения в польскую запись; действия, связанные с оптимальным распределением регистров, выполняются аппаратно.

Другой пример. Традиционно код обращения к полям упакованной структуры представляет собой ряд действий с логическими масками. Причина в том, что обычно в машинах адресация осуществляется с точностью до слова. Поскольку поле — это часть слова, его надо моделировать интерпретационно. В результате код записи в слово памяти отличается от кода записи в поле слова. При наличии условного предложения в левой части оператора присваивания в общем случае надо транслировать динамический выбор того или иного варианта.

Один из встроенных типов МВК Эльбрус — тип имя. С помощью объекта этого типа можно описать не только слово, но и его часть, т. е. поле. Объект подается в универсальную аппаратно реализованную операцию записи; алгоритм операции выбирается динамически, исходя из размера и местоположения адресуемого участка памяти. Это позволяет легко реализовать общую форму левой части оператора присваивания, например случай, когда там находится выбирающее предложение, одна альтернатива которого — простая переменная, другая — элемент массива, а третья — поле.

5. Механизм ситуаций

Прежде чем приступить к описанию механизма ситуаций, объясним причины, по которым обсуждению одного из средств языка посвящен отдельный раздел. Выше в сжатой форме было проведено обоснование основных характеристик языка. Краткость изложения обусловлена тем, что главное назначение книги — дать описание языка. Тем не менее целесообразно, по крайней мере на одном примере, детально отобразить: (1) общую методику исследования, которой придерживались разработчики системы Эльбрус, и (2) конкретное воплощение прин-

ципа совместной разработки языка, аппаратуры и математического обеспечения.

Выбор не случайно пал именно на механизм ситуаций, тем самым достигается и другая цель. Дело в том, что ситуация — это сравнительно новое понятие в развитии современных языков высокого уровня (как станет ясно впоследствии, имеется в виду механизм, отличный от средств обработки прерываний языка PL/1). Варианты подхода предлагаются в материалах по языкам clu [33], ada и в других статьях. В автокоде общий случай механизма ситуаций был введен уже в 1976 г. в первых версиях языка [2, 3]. В программах ОСПО МВК Эльбрус этот механизм используется повсеместно: как средство структурированного программирования, как метод изложения некоторых специфических алгоритмов, как средство обработки ошибок. Например, без преувеличения можно сказать, что вся операционная система МВК Эльбрус запрограммирована без оператора перехода. Ввиду того, что получен достаточно большой опыт эксплуатации, имеет смысл изложить ряд деталей разработки и определить соотношение подхода к решению этого вопроса в автокоде и в других языках.

Ниже, при описании некоторых особенностей механизма ситуаций, проводится сравнение с языком ada. Такой выбор объясняется тем, что, во-первых, этот язык вобрал в себя опыт предшествующих разработок и, во-вторых, материалы по языку содержат достаточно полную информацию.

В связи с этим надо сделать еще одну оговорку. Язык ada в целом рассматривается как определенное продвижение в развитии языков высокого уровня. Общие тенденции языка хорошо согласуются с концепциями системы Эльбрус. Достаточно упомянуть, что авторы языка большое внимание уделяют вопросам, связанным с контекстом обозначений и контекстной защитой. Ряд ограничений языка ada вызван тем, что в соответствии с требованиями, предъявленными к разработке, язык был ориентирован на существующие системы. Учитывая это обстоятельство, приведенное ниже сравнение средств автокода и языка ada целесообразно рассматривать лишь как прием изложения материала, позволяющий лучше оттенить свойства автокода и системы Эльбрус.

5.1. Общий подход. В п. 3.5 уже отмечалось, что наличие и применение оператора перехода объясняется тем, что с практической точки зрения традиционная номенклатура структур управления является неполной. Если исключить из рассмотрения оператор перехода, то оказывается, что семантика каждой фразы описывает только нормальный процесс ее выполнения.

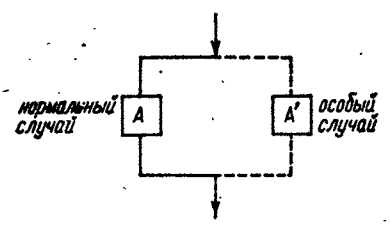
т. е. такой, который завершается при достижении текстуального конца фразы. Например, при выполнении составного оператора алгола-60 последовательно выполняются содержащиеся в нем операторы. В заключение выполняется текстуально последний оператор, что и завершает выполнение составного в целом.

Для обработки особых обстоятельств (см. п. 3.5) желательно наряду с нормальной схемой выполнения иметь схему, в некотором смысле противоположную нормальной. Предположим, что фраза А вложена в В. Вложенность может быть статической (текстуальной) или динамической (в результате вызова процедуры). Предполагаемая схема выполнения должна подчиняться таким требованиям:

1) выполнение А может быть прекращено на некотором промежуточном этапе, до достижения ее текстуального конца;

2) несмотря на прекращение А, управление возвращается в В обычным образом, так, словно А завершилась нормально;

3) алгоритм В может не учитывать, что в А возникли особые обстоятельства. Для этого надо иметь специальную альтернативную конструкцию, с помощью которой можно выразить такую логику: либо А нормально завершается, либо А прекращается, но тогда сразу же выполняется фраза А′, заменяющая А. А′ заменит А в том смысле, что она моделирует эффект, который должна была бы произвести А при нормальном завершении. В обоих случаях управление возвращается в В. Это можно проиллюстрировать такой блок-схемой:



Термин «особые обстоятельства» недостаточно точно отражает суть дела, поскольку в их число могут входить заранее запланированные события, даже такие, отсутствие которых говорит о неправильной работе программы или некорректности входных данных, (см., в частности, пример чтения файла в п. 3.5). Поэтому в автокоде применяется более нейтральный термин — «ситуация». Заменяющую фразу будем называть реакцией на ситуацию.

Перечисленные выше требования подсказывают синтаксические и семантические свойства конструкций, связанных с обра-

боткой ситуации. Во-первых, должно быть средство, позволяющее вводить объекты типа ситуация. Во-вторых, надо уметь сигнализировать о возникновении ситуации. Появление сигнала приводит к прекращению выполнения минимальной охватывающей фразы, после чего выполняется соответствующая реакция. Это соответствие надо каким-то способом обозначать. Непосредственная связь основной фразы и реакции обеспечивается, очевидно, путем их текстуальной конкатенации в рамках единой конструкции.

Участок текста программы, формально являющийся фразой, не всегда выделяется соответствующими ограничителями. Поэтому, во избежание двусмысленностей, целесообразно рассматривать только такие фразы, границы которых явно обозначены. К этому классу относятся так называемые закрытые предложения, каждое из которых ограничено открывающей и закрывающей скобкой (например, **если ... то ... все**, **начало ... конец** и т. д.). При необходимости любая фраза может быть заключена в скобки.

Новая конструкция может иметь такой вид (в примере для конкретности выбран составной оператор, играющий роль основной фразы):

начало ... *s* **!** ; ... **конец** {... реакция ...}

Оператор *s*! сигнализирует о возникновении ситуации *s*. Реакция на ситуацию заключена в фигурные скобки.

Часто бывает недостаточно прекратить выполнение лишь ближайшей охватывающей фразы. Надо уметь отобразить процесс транзитного распространения ситуации в пределах нескольких вложенных фраз. В этом случае реакция представляет собой тривиальное действие, а именно сигнал о том, что данная ситуация возникла в охватывающей фразе.

Пример 7. Распространение ситуации.

```
начало
  ...
  начало
    ...
    s !;
    ...
  конец
  {s !};
  ...
конец
  {... реакция ...}
```

Чтобы не перегружать текст программы, целесообразно при про-

ектировании новой конструкции основываться именно на процессе транзитного распространения ситуации. Для этого надо выделять не ближайшее закрытое предложение, а то предложение, где процесс распространения заканчивается. При этом подразумевается, что выполнение всех вложенных предложений прекращается автоматически.

Соответствующая конструкция автокода называется структурным предложением и имеет вид:

до * *s*1 , * *s*2 , ... , * *sn*
(закрытое предложение)
при
*s*1 : ...реакция на *s*1 ...,
*s*2 : ...реакция на *s*2 ...,
...
sn : ...реакция на *sn* ...
всесит

(о назначении символа * будет сказано ниже). Пример 7 преобразуется к следующему виду:

до * *s*
начало
...
начало
...
s ! ;
...
конец ;
...
конец
при
s : ... реакция ..
всесит

Структурное предложение выполняет две функции:

1) выделяет фразу, на которой заканчивается процесс распространения ситуаций, перечисленных в заголовке. Будем говорить, что заголовок устанавливает «ловушку», перехватывающую распространение ситуаций;

2) задает реакции на ситуации.

Характеризуя свойства структурного предложения в целом, можно сказать, что оно управляет ситуациями.

Конструкцией, сигнализирующей о возникновении ситуации, является структурный переход, имеющий вид:

*s*1

где s — ситуация. Чтобы обеспечить информационную связь между реакцией и точкой программы, где возникла ситуация, естественно воспользоваться традиционным механизмом передачи параметров. Таким образом, в общем случае структурный переход имеет форму:

s!(...список параметров...)

а реакция представляет собой параметризованное действие. В языке ada передача параметров в реакцию невозможна. Это, по-видимому, объясняется сложностью интерпретации этого механизма в условиях отсутствия аппаратно реализованного стека процедур и выражений.

5.2. Поиск ловушки. Рассмотрим, как осуществляется прекращение процесса распространения ситуации. Пусть при выполнении процедуры p может возникнуть ситуация s. Далее предположим, что есть две другие процедуры $p1$ и $p2$, вызывающие p. Чтобы каждая из этих процедур могла прекращать процесс распространения s и самостоятельно реагировать на нее, надо воспользоваться механизмом динамической ассоциации ловушки и ситуации. Процедура $p1$, вызывая p, устанавливает свою ловушку для s. Тогда, в случае возникновения s, процесс ее распространения перехватывается ловушкой из $p1$ и выполняется реакция, определенная в $p1$. Аналогичным образом строится алгоритм $p2$.

Проиллюстрируем динамическую ассоциацию на примере. Пусть *ситлкф* — это ситуация, возникающая в процедуре форматного обмена *читф* при попытке чтения за пределами файла.

Пример 8. Подсчет количества чисел во входном файле.

```
до * ситлкф
(читф (вхфайл, x) ;
   k := k + 1
)
при
ситлкф : запф (выхфайл, k)
всесит
```

В других местах программы при вызове *читф* можно определить иную реакцию на *ситлкф*. Отметим, что динамическая ассоциация ловушки и ситуации не означает нарушения принципа статической известности обозначений. В примере предполагается, что идентификатор *ситлкф* описан в стандартном контексте программы (см. п. 4.6) и поэтому может быть использован в ее тексте.

В процессе выполнения программы может сложиться динамическая иерархия вложенных структурных предложений, управляющих одной и той же ситуацией. Этот случай обрабатывается по магазинному принципу: распространение ситуации перехватывается той ловушкой, которая была установлена позднее всех. Затем эта ловушка ликвидируется и начинает действовать предыдущая. Динамическая ассоциация ловушки и ситуации обеспечивает развязку по переопределению одной и той же ситуации в независимых модулях программы.

Смысл динамической иерархии структурных предложений наиболее просто демонстрируется на примере какого-либо возвратного алгоритма. С этой целью рассмотрим задачу оптимального выбора. Пусть задано n пронумерованных предметов. Каждый предмет характеризуется определенным весом и ценностью. Из этих предметов надо составить набор, общий вес которого не превышает некоторого наперед известного числа, а суммарная ценность — максимальна. Обычно эта задача называется задачей путешественника, потому что при упаковке багажа путешественник руководствуется аналогичными соображениями.

Процесс отбора можно разбить на этапы. На каждом этапе составляется очередной набор, его ценность сравнивается с максимальной из полученных на предыдущих этапах. Следующий этап начинается с того, что из набора, полученного на предыдущем этапе, исключается предмет с максимальным номером k ($k \leqslant n$) и предпринимается попытка добавить предмет с номером $k+1$ и следующие за ним.

Сначала составим программу, не используя механизм ситуаций. Основной алгоритм программируется в виде процедуры *рассмотреть*, которая оценивает возможность добавления в набор i-го предмета и следующих за ним. Для этого предпринимается два шага:

шаг 1: если при добавлении i-го предмета результирующий вес не превышает максимально возможного, то предмет включается в набор; затем рассматривается возможность добавить предмет с номером $i+1$ и следующие за ним;

шаг 2: при исключенном i-м предмете рассматривается возможность добавить предмет $i+1$ и следующие за ним.

В конце каждого этапа (т. е. если предметы исчерпаны) выбирается оптимальный набор.

Чтобы сократить текст примера, ряд несущественных подробностей заменен на комментарии. Кроме того, отсутствует текст вызывающей программы. Предполагается, что в ней описаны объекты, содержащие информацию о текущем наборе,

а также информацию о наилучшем из наборов, полученных на предшествующих этапах. При добавлении или исключении i-го предмета процедура модифицирует данные, касающиеся текущего набора. В конце каждого этапа данные о наилучшем из ранее полученных наборов сравниваются с данными о текущем наборе. Если получен более оптимальный вариант, то происходит соответствующая замена.

Пример 9.а.

```
процедура рассмотреть =проц (i)
    если i <= n то % этап не завершен
        если % добавление i-го возможно
            ...
        то
            ... добавить i-й ... ;
            рассмотреть (i + 1) ;
            ... исключить i-й ...
        все;
        рассмотреть (i + 1)
    иначе % этап завершен
    ... выбрать лучший набор ...
    все
```

Обратим внимание на одну особенность этой программы. После завершения очередного этапа происходит многократный транзитный выход из рекурсивных активаций процедуры *рассмотреть*. Этот процесс заканчивается тогда, когда встретится активация, добавляющая предмет с максимальным номером k (в рамках последнего набора). Этот предмет исключается и начинается следующий этап. Таким способом алгоритм возвращается к k-му предмету. Можно отметить два недостатка этого способа — методологический и технический:

1. Поскольку в процедуре есть два рекурсивных вызова, не очевидно, что при возврате процесс не «проскакивает» нужную активацию.

2. Возврат к k-му предмету моделируется путем многократного выполнения команд выхода из процедуры, что приводит к дополнительным накладным расходам. Желательно иметь механизм, позволяющий выполнить этот возврат за один шаг.

Опишем этот алгоритм, используя механизм ситуаций. Упомянутый процесс транзитного возврата — это, в сущности, процесс распространения ситуации, возникающей при завершении этапа. Обозначим ее идентификатором *этапзавершен*. При добавлении очередного предмета в набор будем устанавливать

ловушку, чтобы процесс распространения ситуации не мог «проскочить» соответствующую активацию.

Пример 9.6.

```
процедура рассмотреть = проц (i)
    если i <= n то % этап не завершен
      если % добавление i-го возможно
         ...
      то
         ... добавить i-й ... ;
         до * этапзавершен % установить ловушку
         (рассмотреть (i + 1))
         при
         этапзавершен : ... исключить i-й ...
         всесит
      все ;
      рассмотреть (i + 1)
    иначе
      ... выбрать лучший набор ... ;
      этапзавершен! % сигнал о возникновении ситуации
    все
```

Возвращаясь к вопросу о программировании возврата к k-му элементу, заметим, что в данной процедуре этот момент вызывает меньше затруднений, чем в предыдущей. Из текста видно, что всякий раз при добавлении предмета устанавливается ловушка, которую невозможно миновать. После того как процесс распространения прекращен, соответствующий предмет исключается. При этом ловушка удаляется автоматически.

Техническое отличие второго примера состоит в том, что здесь не происходит многократный транзитный выход. Структурный переход осуществляет одношаговый возврат в нужную активацию. При этом все промежуточные активации, т. е. такие, которые действительно не имеет смысла рассматривать, будут автоматически ликвидированы.

При соответствующей реализации средств обработки ситуаций эффективность выполнения второй процедуры будет выше. Однако программная интерпретация может сделать обе процедуры либо эквивалентными, либо даже изменить соотношение на обратное. Ниже, при обсуждении реализации механизма ситуаций, мы еще раз вернемся к этому вопросу.

В заключение приведем пример, демонстрирующий создание объекта типа ситуация.

Пример 10. Фрагмент программы, вызывающей процедуру *рассмотреть*.

```
начало
  ...
  конст этапзавершен = ситуация ( );
  процедура рассмотреть = ... ;
  ...
  до * этапзавершен
  (рассмотреть (1))
  при
  этапзавершен : ... печать оптимального набора ...
  всесит;
  ...
конец
```

5.3. Конечная реакция. Остается разобрать случай, когда в процессе распространения ситуации оказывается, что для нее ни одна ловушка не установлена, и в том числе — нет универсальных ловушек. При таких обстоятельствах система должна аварийно прекратить выполнение задачи или процесса. Если нельзя программно определить завершающие действия, то впоследствии трудно будет установить причину ошибки.

Типичный пример — это ошибки, обнаруживаемые при работе системы файлов. В момент возникновения ошибки начинает распространяться некоторая ситуация. Отсутствие ловушки приводит к ликвидации процесса. Ликвидация осуществляется специальной системной процедурой. Предположим, что эта процедура печатает сообщение, содержащее название ситуации. Для этого процедура должна знать все системные ситуации. Номенклатура последних может изменяться. Следовательно, добавление новой ситуации требует модификации этой процедуры. Чтобы сверх того печатать названия ситуаций, описанных в проблемной программе, процедура должна изучить некоторую дополнительную информацию о программе. Как видно, приходим к громоздкому результату. Причина состоит в том, что с самого начала было принято плохо структурированное решение — выполнять в одной общей процедуре конечные реакции на все ситуации.

В автокоде принято другое решение: с каждой ситуацией может быть связана характерная для нее конечная реакция. Последняя представляет собой некоторую процедуру программы, которая задается при генерации объекта типа ситуация. В этой процедуре описываются действия, выполняемые перед аварийным завершением. В частности, одним из действий может быть распечатка сообщения, содержащего название данной ситуации. Информационная связь с местом возникновения си-

туация обеспечивается тем, что процедуре конечной реакции передаются параметры структурного перехода. Возможные причины, по которым этот аппарат отсутствует в языке ada, станут понятны из обсуждения проблем реализации механизма ситуаций.

5.4. Примеры. Обработка ошибок. Остановимся на нескольких примерах применения механизма ситуаций при обработке ошибок.

Обработка прерываний. В первую очередь рассмотрим процесс обработки аппаратно распознаваемых ошибок. К ним относятся такие, как переполнение, выход за границы массива, неверный тип данных, нарушение статуса привилегированности и пр. Для каждой из таких ошибок предусмотрены соответствующие ситуации. Идентификаторы, обозначающие эти ситуации, входят в стандартный контекст программы. При обнаружении ошибки аппаратно запускается процедура обработки возникшего прерывания. Процедура выполняет структурный переход по соответствующей ситуации. Если программа установила «ловушку» на эту ситуацию, то будет выполнена реакция, и работа программы продолжится. В противном случае конечная реакция выдает сообщение и задача аварийно прекращается.

Проиллюстрируем это применение, изменив процедуру примера 1 из п. 4.2 таким образом, чтобы она учитывала возможность переполнения в процессе суммирования:

Пример 11.

```
процедура сумма = функция (в)
    до * переполнвещ
    начало
        ф64 с := 0 ;
        для k до (длина в — 1) % цикл от k = 0
        цикл
            с := с + в [k]
        повторить ;
        в
    конец
    при переполнвещ : максвещ % максимальное число
    всесит
```

Стандартная ситуация *переполнвещ* возникает при переполнении результата арифметических операций. В данном случае, если очередная операция сложения приведет к переполнению, то выполнение процедуры будет прекращено; в качестве результата выдается максимально возможное вещественное число *максвещ*. В противном случае процедура выполняется обычным образом.

Ошибки обмена. Аналогичным образом обрабатываются ошибки взаимодействия программы с системой, в частности, ошибки обмена. В общем контексте описаны идентификаторы ситуаций для таких ошибок, как логический конец файла, конец ленты, ошибка символа, нет ресурса требуемого типа, нет имени в справочнике и пр. Стандартная реакция, осуществляемая подсистемой файлов при обнаружении ошибки, состоит в том, что выполняется структурный переход по соответствующей ситуации. Программа может перехватить процесс распространения ситуаций и предпринять необходимые действия. Пример обработки приведен в п. 5.2 (пример 8).

Контроль интерфейса. При взаимодействии с системой часто встречаются ошибки, причиной которых является несоблюдение интерфейса, например, неверный тип, число или значения параметров, неправильное соотношение между значениями параметров и пр. (отметим, что статический контроль охватывает лишь незначительный диапазон подобных ошибок). Ошибка в исходных данных может в последующем привести к прерыванию в отлаженной системной процедуре. В диагностическом сообщении такое обстоятельство нужно квалифицировать как ошибку в интерфейсе, а не как ошибку в системной процедуре. Это можно осуществить, введя в процедуру безусловный контроль параметров на входе. Здесь, однако, надо учесть, что появляются дополнительные накладные расходы, связанные с безусловным контролем.

В системе Эльбрус можно применить другой метод, а именно — метод, использующий механизм ситуаций. Для этого системная процедура, доступная пользователю, устанавливает так называемую «универсальную ловушку» на любые ситуации, которые могут возникнуть в процессе работы. Реакция должна предусматривать контроль интерфейса, анализ текущей обстановки и остальные действия, гарантирующие нормальное функционирование системы и, по возможности, полную диагностику ошибки. В результате, контроль будет выполняться не безусловно, а только в том случае, если интерфейс действительно был нарушен.

Аналогичный метод могут использовать и пакеты прикладных программ. Реакция на ошибку должна содержать анализ параметров и структурный переход по программно определяемой ситуации, сигнализирующей о нарушении интерфейсных соглашений. Эта ситуация (одна или несколько) описывается вместе со всеми процедурами пакета. Вызывающая программа транслируется в контексте пакета, куда включены не только идентификаторы его процедур, но и идентификаторы аварий-

ных ситуаций. Следовательно, эти идентификаторы известны вызывающей программе, и она может установить ловушки и определить соответствующие реакции.

5.5. Реализация. Рассмотрим два наиболее существенных аспекта реализации механизма ситуаций: способ установки и поиска ловушек и метод моделирования процесса распространения ситуации.

Идентификация ситуации. Для реализации средств обработки ситуаций в первую очередь необходимо разработать механизм, с помощью которого осуществляется ассоциация ситуации и ловушки. В качестве исходной модели можно принять механизм именования переменных. Действительно, ситуацию можно рассматривать как имя переменной, а ловушку — как пару из имени переменной и ее значения; значением является метка кода реакции. Чтобы реализовать динамическую ассоциацию, надо: (1) каким-либо образом организовать стек поколений ловушек и (2) обеспечить уникальность имен. Тогда ассоциативный поиск позволяет находить ближайшую ловушку с данным именем.

До этого момента ход рассуждений достаточно очевиден и совпадает как в автокоде, так и в языке ada. Далее наблюдается существенное расхождение в том, какими способами организуется стек поколений и обеспечивается различимость имен. В языке ada ситуация не является объектом определенного типа, а, наподобие метки перехода, рассматривается как некое специфическое образование, для которого возможно только статическое описание. В качестве имени предлагается выбрать целое число, назначаемое во время трансляции. При этом отмечается, что локальные ситуации рекурсивных процедур становятся, пользуясь терминологией алгола, собственными величинами.

Обратим внимание на другой недостаток такого метода реализации. Пусть процедура $a1$ пакета A вызывает процедуру $b1$ пакета B, которая в свою очередь вызывает процедуру $a2$ пакета A. В процедуре $a2$ возникает ситуация x, которая известна в контексте $a1$ и $a2$, но неизвестна в контексте $b1$. Если пакеты A и B были оттранслированы независимо и перед исполнением не прошли редакцию связей, то возможно случайное совпадение числа, назначенного для x, и числа, назначенного для какой-либо локальной ситуации процедуры $b1$. В результате может возникнуть трудно диагностируемая ошибка. Можно пронумеровать все пакеты в системе и идентифицировать ситуацию парой чисел: номер пакета, номер ситуации. Тем самым в системе вводится специальное адресное пространство

ситуаций. Такой подход, хотя и возможен, но представляется достаточно громоздким.

Существует другое решение: все динамически взаимодействующие программы перед началом выполнения должны пройти этап статического редактирования связей. Следовательно, динамический вызов независимо транслированной программы исключается. На практике такое ограничение, по-видимому, неприемлемо. Например, взаимодействие программы пользователя и системной программы, взаимодействие диалоговой системы и программы пользователя происходит с помощью динамического вызова. Нецелесообразно исходить из того, что все программы в системе скомплексированы. Иными словами, рассматриваемый метод реализации затрудняет использование языка в качестве единого средства программирования системных и проблемных программ.

Универсальный метод — это динамическое назначение имен. Например, в процессе выполнения программы с каждой новой ситуацией связывается число, на единицу большее. Тогда все существующие в данный момент ситуации имеют различные имена. Это, в частности, решает проблему локальных ситуаций рекурсивных процедур.

Назначая ситуациям отличающиеся целые числа, мы, по существу, ввели новое адресное пространство. Вместо этого естественно использовать существующую математическую память, т. е. для каждой новой ситуации надо отводить ячейку, а ее адрес использовать в качестве имени. Это хорошо согласуется и с тем, что надо обеспечить конечную реакцию, так как процедуру реакции можно хранить именно в этой ячейке.

Таким образом, анализируя другие методы реализации, приходим к решению, которое в автокоде было положено в основу разработки. Ситуация трактуется как составной объект определенного типа. Для этого объекта, как и для других, существуют средства динамической генерации. Указатель ситуации может являться значением переменной, константы или параметра. Кроме того, для него определены действия: структурный переход и формирование ловушки. Единственным атрибутом этого объекта является процедура конечной реакции.

Структурный переход. Полностью интерпретационный метод реализации механизма процедур и ситуаций связан со значительными накладными расходами. Процесс распространения ситуации моделируется путем вызова специальной административной процедуры, которая считывает подряд все элементы списка связующей информации активаций процедур, считывает

коды соответствующих процедур и производит поиск ловушки по списку, находящемуся в коде.

В процедуру может быть вложено несколько пересекающихся или непересекающихся блоков, управляющих одной и той же ситуацией. Поэтому поиск действующей ловушки по статическому списку — это трудоемкий процесс.

Можно попытаться организовать стек поколений ловушек в той же области, где расположены локальные данные активаций процедур, т. е. в общем стеке. Здесь, однако, начинает играть роль бестиповое хранение информации в памяти: ловушки невозможно обнаружить путем анализа содержимого ячеек локальной области. Следовательно, их надо связывать в список и динамически осуществлять коррекцию этого списка при входе в блок с ловушкой и при выходе из него. В процессе выполнения программы всегда будет тратиться время на коррекцию списка независимо от того, возникнет ситуация или нет. С другой стороны, естественно потребовать от метода реализации, чтобы основные расходы приходились на обработку структурного перехода, а не на превентивную установку ловушек. В целом приходим к выводу, что без соответствующей аппаратной поддержки трудно найти достаточно эффективный метод реализации механизма ситуаций.

В аппаратуре МВК Эльбрус понятие ситуации отображается следующим образом:

1) в момент установления ловушки связующая информация активации текущей процедуры специальным образом метится;

2) объект, идентифицирующий ситуацию (указатель ситуации), имеет аппаратно распознаваемый тип;

3) команда «безусловный переход» играет роль универсальной операции, выбирающей алгоритм выполнения в зависимости от типа операнда. Если операндом является указатель ситуации, то эта команда отыскивает в динамической цепочке связующей информации активаций процедур ближайшую помеченную активацию. Тем самым моделируется процесс распространения ситуации. Затем вызывается системная процедура обработки ситуаций.

Хранение информации вместе с типом позволяет поместить ловушки в общий стек, не связывая их в список. Ловушка занимает две ячейки: в одной находится указатель ситуации, в другой — метка перехода на тело реакции. Действия, связанные с установкой или уничтожением ловушки, равносильны двум присваиваниям, т. е. можно считать, что здесь расходы незначительны.

После того как аппаратно локализована ближайшая помеченная активация, системная процедура, имея указатель ситуации, ищет ловушку с таким же указателем. Это действие выполняется с помощью специальной операции ассоциативного поиска, реализованной в аппаратуре. Если ловушка найдена, то осуществляется переход на код реакции. Здесь, в первую очередь, производится перепись параметров из верхней части стека в отведенные для них ячейки локальной области; затем начинает выполняться тело реакции. Если ловушка не найдена, то процедура убирает пометку из связующей информации и выполняет структурный переход, моделируя тем самым дальнейшее распространение ситуации *).

Таким образом, накладные расходы на реализацию механизма ситуаций сбалансированы так, чтобы основная доля приходилась на обработку процесса распространения ситуации. Аппаратная реализация позволяет: (1) избежать обособленной трактовки ситуаций; в автокоде они вводятся как обычные объекты одного из возможных типов; (2) уменьшить объем интерпретации и упростить алгоритмы интерпретации; (3) ликвидировать необходимость редактирования связей.

При разработке языка и системы учитывалось, что низкая эффективность реализации тех или иных конструкций приводит к тому, что они не используются в практическом программировании. Следствием больших расходов на интерпретацию механизма ситуаций может быть возврат к традиционным методам программирования с оператором перехода. Обратимся к примеру 9.б. Только что описанный метод позволяет аппаратно за один шаг локализовать активацию процедуры *рассмотреть*, соответствующую последнему предмету, включенному в набор. При этом автоматически минуются все промежуточные активации. Интерпретационный метод производит работу, близкую к той, которая описана в примере 9.а. Более того,

*) Здесь мы встречаем пример, когда ситуация используется не только для передачи управления, но и как объект обработки. Во-первых, ситуация передана системной процедуре в качестве параметра. Далее, в процессе обработки процедура присваивает ситуацию локальным переменным, использует в операциях сравнения и других конструкциях общего назначения. Если же ситуации придана обособленная трактовка (например,— разрешено использовать ее только в структурном переходе, как это сделано в языке ada), то описать алгоритм средствами языка достаточно сложно.

если не метить связующую информацию, то может оказаться, что эффективность процедуры, не использующей механизм ситуаций, выше.

Эффективная реализация дает возможность расширить область применения механизма ситуаций и рассматривать их не только как средство обработки ошибок, но и как инструмент, позволяющий по-иному осмыслить многие алгоритмы и запрограммировать их более адекватным и очевидным способом. Заметим, что в ряде приложений потеря эффективности не приемлема даже при обработке ошибок. Это, например, относится к программам реального масштаба времени, где требуется быстрая реакция на ошибки.

5.6. Статические ситуации. Рассмотрим частный случай общего механизма ситуаций. Пусть для данной ситуации статически известно, что есть только одно управляющее ею структурное предложение. Это условие выполняется при одноуровневом возврате из процедуры, при выходе из статической иерархии вложенных циклов и в других алгоритмах, где область распространения ситуации статически детерминирована.

При разработке автокода этот случай выделялся особо. Во-первых, он достаточно типичен, и, во-вторых, его можно реализовать с помощью безусловного перехода на метку, не вовлекая общий механизм ловушек. Попытка ввести оптимизацию, основанную на статическом выяснении взаимодействия конструкций, связанных с обработкой ситуаций, приводит к усложнению структуры транслятора. Кроме того, необходимо учесть, что модификация текста программы может повлечь за собой изменение статической картины связей, что сделает невозможной частичную перетрансляцию модифицированного фрагмента. Наконец, во многих приложениях требуется, чтобы язык не скрывал объективно существующей разницы в реализации простых и сложных понятий. На основе этих соображений в автокоде были введены два класса ситуаций: динамические и статические. Динамические ситуации — это общий случай, а статические — рассматриваемый здесь частный случай.

Особенность статических ситуаций заключается в том, что они вводятся не с помощью динамического генератора, а описываются в заголовке структурного предложения. Это же предложение и управляет ситуацией, например:

```
до ss
начало
    ...
    ss !!
    ...
```

```
конец
при
    ss : ...
всесит
```

В заголовке отсутствует символ * перед идентификатором. Это означает, что *ss* — это идентификатор новой статической ситуации, а не обозначение ранее созданной динамической (для сравнения см. пример 10).

Здесь есть прямая аналогия с появлением управляющей переменной в заголовке цикла типа арифметической прогрессии. В автокоде, как и в алголе-68, это вхождение одновременно служит описанием управляющей переменной. Тем самым достигаются три цели: упрощается программирование конструкции; подчеркивается, что идентификатор связан лишь с одной фразой; обеспечивается оптимальная трансляция. Как видно, такой же прием применен и для статических ситуаций. В остальном обработка статических и динамических ситуаций описывается идентичными конструкциями.

Поскольку в языке ada предложено иное решение проблемы, необходимо провести сравнение. В предварительном сообщении описаны дополнительные средства управления: цикл-while, условный оператор выхода из цикла, оператор выхода из процедуры (возможно, с выдачей значения). Такой набор тоже позволяет обеспечить эффективность обработки простых случаев. Кроме того, преследовались две другие цели: повысить выразительность текста за счет расширения номенклатуры конструкций и ввести оператор, выдающий значение процедуры функции.

Такой же подход был опробован в ранних версиях автокода. Опыт эксплуатации показал, что минимизация номенклатуры конструкций, связанных с передачей управления, облегчает использование языка и упрощает изучение программ. Выразительность текста достигается тем, что: (1) описание идентификатора статической ситуации текстуально выделяет ту фразу, выполнение которой может прекратиться; (2) содержательный смысл этого идентификатора можно рассматривать, как абстрактное обозначение условия прекращения; (3) восклицательный знак позволяет легко отыскать в тексте программы структурный переход и конкретизацию условия прекращения. Например, смысл цикла:

```
до приближение найдено
цикл
  ...
```

```
    если ак < эпсилон то приближениенайдено ! все}
    ...
повторить
```

уже частично прокомментирован непосредственно конструкциями языка. Тот факт, что опущена реакция на ситуацию *приближениенайдено*, означает, что она выбирается по умолчанию. В данном случае умолчание — пустой оператор.

Параметризация структурного перехода позволила не вводить в автокод специальный оператор выхода из процедуры-функции, а ограничиться статическими ситуациями. В типичном случае процедура-функция программируется следующим образом:

```
процедура генмассив = функция (...)
    до памятьвыделена
    начало
        ...
        памятьвыделена ! (в) }
        ...
    конец
```

Умолчание для данного случая состоит в том, что значение выражения *в* выдается в качестве результата функции.

В целом перечисленные соображения позволили в последней версии автокода ликвидировать дополнительные средства передачи управления. В настоящий момент в язык введены:

— традиционные структуры управления (последовательность операторов, условное и выбирающее предложения, цикл);

— структурное предложение и структурный переход.

6. Заключение. Универсальность языка

Возвращаясь к языку в целом, сделаем выводы относительно области его применения:

1. Автокод можно использовать в качестве обычного проблемного языка высокого уровня.

2. Возможности автокода позволяют использовать его для программирования больших систем, предъявляющих повышенные требования к языкам:

— трансляторов, редакторов и других систем обработки текстовой информации;

— диалоговых и интерпретирующих систем, планировщиков систем построения пакетов прикладных программ, систем управления базами данных и других программных комплексов, обрабатывающих объекты (оперативные и внешние), число

которых, типы и объем занимаемых ресурсов статически неизвестны; при обработке внешних объектов могут быть статически неизвестны и другие характеристики: архивы, где находятся обрабатываемые объекты (в частности, программы), архивные обозначения объектов.

3. Автокод можно использовать для программирования таких сугубо системных задач, как распределение ресурсов вычислительного комплекса (процессоры, оперативная и внешняя память). Подобное применение обусловлено высоким уровнем архитектуры и системы команд МВК Эльбрус: оставаясь в рамках понятий ЯВУ, можно получить доступ ко всем аппаратным возможностям. Это позволило не разрабатывать в системе Эльбрус специальный язык системного программирования. В настоящее время на автокоде реализовано все общее системное программное обеспечение МВК Эльбрус: операционная система, включающая подсистему файлов, трансляторы автокода и других ЯВУ, остальные компоненты системы программирования.

4. Одним из следствий расширения возможностей языка является и то, что он используется в качестве языка управления заданиями. Хочется подчеркнуть, что подобное применение автокода не являлось самостоятельной целью разработки. Скорее — это одно из частных следствий общих решений. Решающую роль сыграло введение в язык внешнего объекта, как объекта обработки, во многом равноправного с оперативным.

В целом можно сказать, что характеристики автокода позволяют рассматривать его как единое средство общения программиста с вычислительной системой, т. е. как универсальный язык программирования.

Наряду с универсальностью автокод обладает тем свойством, что он позволяет получать эффективные и надежные программы. Первое свойство обеспечивается эквивалентностью понятий языка и системы. Второе свойство — надежность — основывается на описанных выше механизмах защиты и контроля. Наличие в языке развитых средств обработки ошибок также играет важную роль в обеспечении надежности программ.

В заключение отметим, что универсальность, эффективность и надежность в данном случае не противоречат друг другу. Скорее можно сказать, что обобщенные характеристики автокода вполне согласованно вытекают из основной идеи разработки: расширить сферу применения концепций и механизмов языков высокого уровня и в достаточно общем виде реализовать эти механизмы средствами аппаратуры и операционной системы.

ГЛАВА 2
БАЗОВЫЕ КОНСТРУКЦИИ

1. Основные понятия

Текст программы содержит описание алгоритма обработки объектов. В процессе выполнения алгоритма могут предприниматься следующие основные действия: арифметические, логические и другие операции над объектами, присваивание, выполняющее замену текущего значения переменной, создание и уничтожение объектов. К числу основных действий следует отнести также операции над внешними объектами, в частности, ввод/вывод и поиск в архиве.

Структурирование текста и действий программы производится с помощью процедур и закрытых предложений. Процедура может иметь параметры, обеспечивающие ее информационную связь с другими частями программы. При вызове процедуры происходит замена формальных параметров на фактические, после чего управление передается вызываемой процедуре.

В число закрытых предложений входят: условное и выбирающее предложения, цикл и структурное предложение. Первые два выбирают одну из заключенных внутри них альтернатив и передают ей управление. Каждая из альтернатив является последовательным предложением, задающим прямую (текстуальную) последовательность действий. Цикл организует повторное выполнение последовательного предложения. Структурное предложение и связанный с ним механизм ситуаций позволяет при заданных обстоятельствах прекращать выполнение закрытого предложения еще до того, как будет достигнут текстуальный конец последнего. Это средство используется и для обработки исключительных обстоятельств (например, ошибок), и для описания обычных алгоритмов. Например, структурное предложение может быть использовано для программирования окончания цикла.

1.1. Объекты, переменные и константы. Объект характеризуется типом, определяющим набор допустимых операций над

объектом. Типы разбиваются на два класса: типы простых объектов и типы составных объектов (см. пп. 1.4 и 1.5). На рис. 2.1 приведена полная номенклатура объектов, с которыми может оперировать программа.

Количество разрядов, занимаемое простым объектом, называется форматом этого объекта. Например, вещественное число

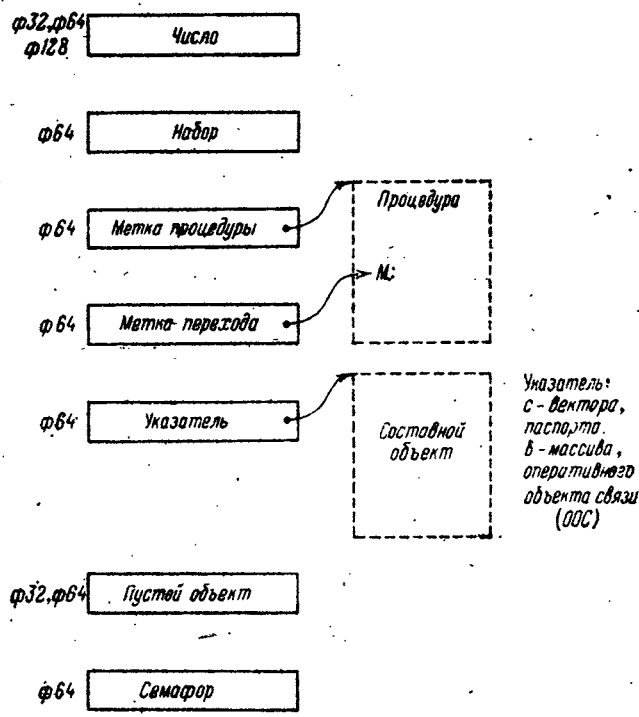


Рис. 2.1, *а*. Номенклатура объектов языка, простые объекты.

может занимать 32, 64 или 128 разрядов (от формата числа зависит точность его представления, а также максимально и минимально возможная абсолютная величина).

Переменная имеет две характеристики *): формат и текущее значение. Формат определяет размер отводимой для нее обла-

*) В языке нельзя явным образом оперировать с объектами типа имя (см. § 2 гл. 1). Поэтому для простоты термин «имя переменной» в описании языка не используется. Заметим, что косвенное обращение к переменной, тем не менее, возможно (см. п. 6.4).

сти памяти. Значением переменной может быть только простой объект. Переменная не имеет определенного типа — одна и та же переменная может в разное время принимать значения различных типов.

Отличие константы от переменной состоит в том, что значение константы неизменно в процессе выполнения программы.

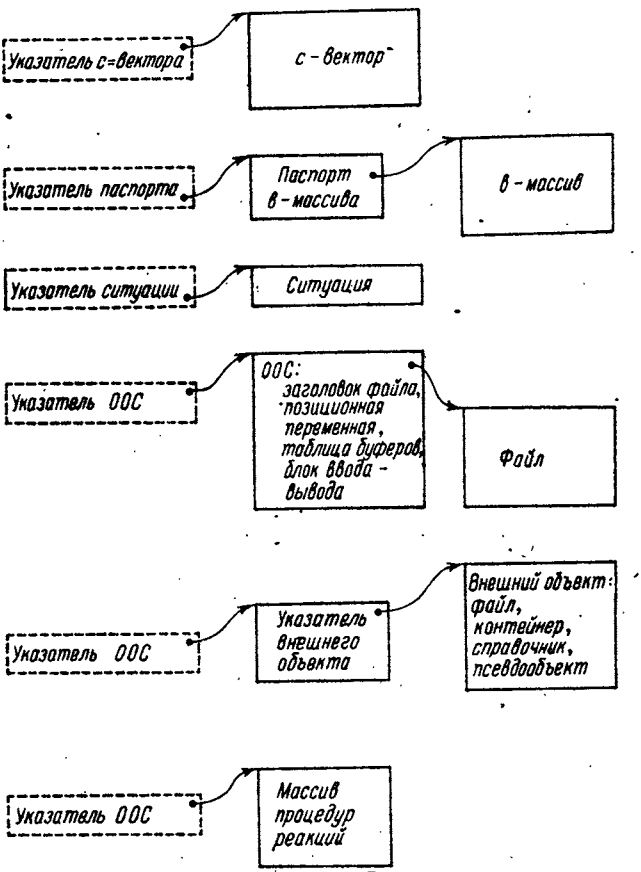


Рис. 2.1, *б*. Номенклатура объектов языка, составные объекты.

Существуют константы статического класса и константы динамического класса. Разница между ними в том, что для констант первого класса память не отводится (они содержатся в коде программы), а для констант второго класса — отводится,

1.2. Обозначения и конструкции. Текст программы строится из конструкций: описаний и предложений.

Описание вводит константу или переменную и связывает с ней обозначение (идентификатор).

Предложение может обозначать:

— объект, явно заданный своим изображением, например: 3.14 (число), истина (логическое значение).

— действия над объектами, например $x + y$. Ряд предложений, обозначающих действия, обладает тем свойством, что в результате их выполнения получается некоторый объект. Например, результат выполнения $x + y$ есть некоторое число. Будем говорить, что такие предложения выдают значение. Этим значением является результирующий объект. Его для краткости будем называть значением предложения.

— переменную или константу, например: x — идентификатор переменной или константы, $a[i]$ — обозначение элемента массива (последний является переменной или константой, см. п. 6.2). В состав подобных обозначений могут входить обозначения для действий, например: $i+1$ в предложении $a[i+1]$. При этом предполагается, что сначала выполняются действия, а затем определяется обозначаемая переменная или константа.

— последовательность, в которой выполняются действия, например:

если q **то** $x := x + y$ **иначе** $a\ [i] := 3.14$ **все**

1.3. Форматы. Форматы бывают стандартные и нестандартные. Стандартные форматы — это следующие: бит — 1 разряд (ф1), тетрада — 4 разряда (ф4), байтовый или литерный формат — 8 разрядов (ф8), половина слова — 32 разряда (ф32), слово — 64 разряда (ф64), два слова — 128 разрядов (ф128). Первые три формата называются строчными, остальные — простыми. Относительно нестандартных форматов см. п. 1.4.

1.4. Простые объекты. Различаются следующие типы объектов*): целое, вещественное, набор, метка процедуры, метка перехода, несколько типов указателей, семафор и тип пустой объект, обозначающий неинициализированные данные. Целое число может иметь формат ф32 или ф64, вещественное — ф32, ф64 или ф128. Набор, метка и указатель имеют формат ф64. Пустой объект может иметь формат ф32 или ф64.

При выборке объект распаковывается. Представления объектов в упакованном и распакованном виде различается для

*) Здесь и далее в этом пункте под термином «объект» подразумевается простой объект.

форматов ф1, ф4, ф8, ф32. Внутреннее распакованное (ВРП) и упакованное представления объектов приведены в Приложении 2.

Целое. Объекты типа целое представляют собой целые числа со знаком. Максимальная абсолютная величина целого формата ф32 имеет порядок $2*10^9$, а целых формата ф64 — $9*10^{18}$. Точные цифры приведены в п. 3.1.

Вещественное. Абсолютная величина вещественных чисел заключается в следующих диапазонах:

вещественное ф32 и ф64: $9*10^{-78} \leqslant x \leqslant 7*10^{75}$,

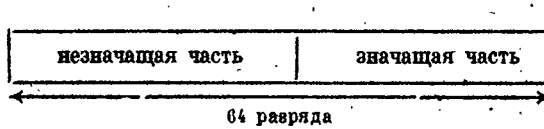
вещественное ф128: $8*10^{-19728} \leqslant x \leqslant 1*10^{19727}$

(в п. 3.1 приведены более точные цифры).

В формате ф64 представляется тот же диапазон чисел, что и в формате ф32, но с большей точностью: 8 десятичных цифр мантиссы в формате ф32 и 17 цифр в формате ф64. По сравнению с форматом ф64, в формате ф128 можно представить больший диапазон чисел с большей точностью (до 34 десятичных цифр мантиссы).

* Внутреннее представление вещественных чисел — шестнадцатеричное. Абсолютная величина числа определяется формулой $m*16^p$, где m — дробная мантисса (точка перед старшим разрядом), а p — целый двоичный порядок. Мантисса нормализована, т. е. старшая тетрада отлична от нуля. Вещественный нуль, являющийся минимальным по абсолютной величине представимым числом, имеет максимальный отрицательный порядок. *

Набор. Набор — это 64-разрядный объект, состоящий из значащей и незначащей частей:



Значащая часть прижата вправо. Под термином «длина набора» понимается длина его значащей части. Всюду, где не оговорено особо, предполагается, что в операции участвуют все 64 разряда, причем незначащая часть заполнена нулями.

Набор имеет несколько интерпретаций:

а) В ряде предложений набор рассматривается как последовательность элементов того или иного строчного формата. В этих случаях набор играет роль одного из следующих шести типов:

1. Битовое, т. е. последовательность элементов формата ф1.

2. Цифровое, т. е. последовательность элементов формата ф4, заполненных шестнадцатеричными цифрами.

3. Литерное, т. е. последовательность элементов формата ф8, заполненных либо кодами литер, либо целыми, величина которых заключается в диапазоне от 0 до 255.

4. Логическое, т. е. истина или ложь.

5. Цифра, т. е. один элемент формата ф4.

6. Литера, т. е. есть один элемент формата ф8.

Элементы набора нумеруются справа налево, начиная от нуля.

* Если длина набора не кратна подразумеваемому формату элемента, то значащая часть дополняется слева нулями до ближайшей кратной длины. В операциях, требующих одноэлементных наборов, принимается во внимание только правый крайний элемент. *

б) Предложения, определенные для целочисленных операндов, интерпретируют набор как целое без знака (при этом считается, что значащая часть набора содержит двоичное представление целого). Тем самым обеспечивается автоматическое преобразование из битовых, цифровых и литерных в целое. В дальнейшем, если иное не оговорено, подразумевается, что целочисленный операнд всегда может быть представлен набором. Величина эквивалентного целого не должна превышать $2^{63} - 1$.

* В случае, если старший разряд исходного полного набора, т. е. разряд, соответствующий знаку целого числа, равен 1, упомянутое преобразование приводит к возникновению ситуации *неверныйоперанд*. *

в) Еще одной областью применения наборов являются вычисления, связанные с обработкой упакованной информации (см. ниже).

Метка процедуры. Метка процедуры — это объект, используемый для доступа к процедуре. Она содержит информацию о коде процедуры и о контексте идентификаторов (см. § 4), в котором ее следует выполнять. Далее для краткости вместо термина «метка процедуры» будем применять термин «процедура».

* *Метка перехода*. Этот объект используется при передаче управления как указатель помеченного предложения программы. *

Пустой объект. Этот объект в основном используется для представления начального значения неинициализированных переменных простого формата. Кроме того, его можно присваи-

вать и передавать как параметр; он допускается в операциях проверки типа. В остальных случаях использование этого объекта приводит к ошибке.

* Ошибка, связанная с использованием пустого объекта, приводит к возникновению ситуации *неверныйоперанд*. Ошибка возникает при выполнении операций над таким объектом, но не в момент выборки его из переменной. *

Указатели. Доступ к составным объектам осуществляется посредством указателей. Различаются следующие типы указателей: указатель смежного вектора, указатель паспорта выстроенного массива, указатель оперативного объекта связи с внешним объектом. Конкретное назначение каждого типа рассматривается в п. 1.5. В дальнейшем будем говорить, что указатель приводит к составному объекту.

Упаковка информации в полях. Объект, формат которого не превышает 64 разрядов, можно хранить в упакованном и распакованном виде. Таким объектом может являться целое, одно- или многоэлементное битовое, цифровое или литерное. В упакованном виде объект может содержаться в поле вектора, состоящего из элементов строчного формата, или в поле переменной. Полем вектора называется последовательность нескольких смежных элементов вектора, а полем переменной — часть памяти, занимаемой переменной. В языке существуют средства выборки из поля и присваивания полю. При выполнении присваивания объект обрезается слева до длины поля; оставшиеся разряды помещаются в поле. При выборке происходит распаковка; в качестве распакованного представления используется набор, в котором содержимое исходного поля является значащей частью. Таким образом, тип результата выборки не зависит от типа объекта, помещенного в поле. Это, однако, не мешает упаковывать в полях не только наборы, но и целые, так как операции и конструкции, требующие целочисленных операндов, воспринимают набор, как целое без знака. Таким образом, после выборки целочисленного значения из поля его не надо преобразовывать к типу целое: это преобразование автоматически делается в операциях.

* Допускается упакованное хранение целых со знаком. С этой целью в особых случаях при выполнении присваивания полю происходит перенос знакового разряда исходного целого в старший разряд поля (см. § 8). Чтобы результат выборки такого поля можно было использовать в операциях, его надо предварительно преобразовать в целое с помощью операции целзн.

Средства работы с полями предназначены также для программирования алгоритмов, анализирующих и формирующих объекты во внутреннем представлении.

Семафор. Объект типа семафор используется при синхронизации параллельных процессов. *

1.5. Составные объекты. Различаются следующие встроенные типы составных объектов: смежный вектор, выстроенный массив, паспорт выстроенного массива, динамическая ситуация, один из типов оперативных объектов, предназначенных для связи с внешними объектами, один из типов внешних объектов. Объект, отличный от внешнего, называется оперативным.

Составной объект может иметь ряд атрибутов, характеризующих особенности структуры информации, содержащейся в нем, и особенности выполнения операций над объектом.

Массив. Массив представляет собой совокупность элементов (переменных или констант) одинакового формата. С каждым элементом связан один или несколько индексов, однозначно определяющих элемент. Длина массива вычисляется при его создании или объявляется плавающей. В последнем случае ее можно изменять в процессе работы. Максимально возможное колнчество элементов массива зависит от формата элемента:

ф128	ф64	ф32	ф8	ф4	ф1
$2^{20}-1$	$2^{20}-1$	$2^{20}-1$	$2^{19}-1$	$2^{18}-1$	$2^{16}-1$

Массив может иметь тип смежный вектор (с-вектор) или выстроенный массив (в-массив).

Смежный вектор. Элементы такого вектора располагаются в памяти смежно. Для доступа к с-вектору используется указатель с-вектора.

Выстроенный массив может содержать n измерений ($n \geqslant 1$). Элементы в-массива расположены в памяти так, что смещение элемента относительно начала массива линейно зависит от его индексов. Отсюда название «выстроенный массив». Доступ к в-массиву осуществляется посредством промежуточного объекта типа паспорт выстроенного массива. Доступ к паспорту также происходит косвенно посредством указателя паспорта. Таким образом, непосредственным представителем в-массива в программе является указатель паспорта. Создание массива сопровождается неявным созданием паспорта и указателя паспорта.

Отличие одномерного в-массива (в-вектора) от с-вектора проявляется в следующем. С точки зрения использования с-вектор представляет больше возможностей, чем выстроенный

вектор. Например, существует ряд групповых операций, определенных только для с-векторов: пересылка, поиск элемента, сравнение. С другой стороны, в-вектор реализует общий случай представления в памяти: элементы вектора могут располагаться несмежно на одинаковом расстоянии друг от друга. Такую структуру имеет одномерное сечение многомерного массива.

Ситуация. Ситуации — это средство идентификации обстоятельств, при которых происходит прекращение выполнения закрытых предложений (см. § 1). С целью оптимизации различаются статические и динамические ситуации. В первом случае ситуация и соответствующая реакция задаются статически. Во втором случае ситуация создается генератором, и реакцию можно динамически переопределять.

* В процессе выполнения операций, встроенных в язык, могут быть обнаружены ошибки, например, переполнение результата арифметической операции или попытка чтения за границей файла. Для подобных случаев предусмотрены стандартные ситуации. Считается, что в стандартном контексте программ описаны константы, значениями которых являются эти ситуации. В описании семантики операций указываются обстоятельства, при которых возникают ситуации, и приводятся идентификаторы соответствующих констант. Поскольку идентификаторы известны, программа может задать собственную реакцию на ошибочную ситуацию (подробнее см. § 10).

Ниже, в описании семантики конструкций, приводятся ограничения на возможные типы значений исходных операндов и компонентов. Если при выполнении программы эти ограничения нарушаются и иное не оговорено, то подразумевается, что возникает ситуация *неверныйоперанд*.

Статическая ситуация — это константа, значением которой является метка перехода. Динамическая ситуация — это составной объект, доступ к которому осуществляется посредством указателя с-вектора. *

Внешние объекты. Внешний объект — это объект типа файл, контейнер или справочник. Файл содержит данные или псевдообъект (программу или вложенный файл). Контейнер содержит совокупность файлов. Справочник предназначен для организации архива. Взаимодействие программы с внешним объектом осуществляется посредством оперативных объектов связи.

Оперативные объекты связи. Существуют следующие типы оперативных объектов связи:

— указатель внешнего объекта, используемый при поиске по архивным справочникам;

— заголовок открытого файла, заголовок открытого контейнера и блок ввода/вывода, предназначенные для работы с объектами в открытом состоянии;

— позиционная переменная и таблица буферов, используемые в процессе буферизованного обмена с файлом;

— объект типа массив процедур реакций, используемый при управлении обработкой ошибок, возникающих в процессе взаимодействия с внешним объектом.

Все перечисленные объекты создаются динамически. Доступ к ним осуществляется с помощью значений типа указатель оперативного объекта связи.

1.6. **Динамическое распределение памяти, локализация.** Память, необходимая для переменных, констант динамического класса и составных объектов, отводится и освобождается динамически. Время, в течение которого память занята, связано с временем выполнения единиц программы, называемых блоками. Блоком является процедура, закрытое предложение, начинающееся с описаний, и последовательное предложение, начинающееся с описаний.

Переменные и константы динамического класса, вводимые с помощью описаний, являются локальными в том смысле, что необходимая память отводится при выполнении описания и освобождается по окончании выполнения соответствующего блока.

Память для составных объектов отводится при выполнении генератора объекта. Оперативный объект может быть локальным или глобальным. Локальный объект существует до конца выполнения блока, в котором он создан. Глобальный объект существует до конца задачи.

Внешний объект может быть локальным, глобальным или постоянным. Объект называется постоянным в том случае, если есть другие внешние объекты, содержащие ссылки на него. Время жизни локального и глобального внешнего объекта определяется, как указано выше. Постоянный объект существует до тех пор, пока существуют ссылки на него. Для объектов всех типов существуют средства явного уничтожения.

1.7. **Статус привилегированности.** Процедура может быть привилегированной и непривилегированной. В непривилегированных процедурах запрещены привилегированные действия: изменение поля адресного объекта (указатель, метка)*), изменение типа неадресного объекта на какой-либо адресный тип. При нарушении этих ограничений возникает ситуация *приведействия*.

Статус процедуры определяется статусом программы. Привилегированная программа может содержать процедуры обеих разновидностей, а непривилегированная — только непривилегированные процедуры. Аналогичным образом статус программы зависит от статуса пользователя. Статус пользователя назначается при регистрации его в системе. Необходимо отметить,

*) Введение статуса привилегированности — это технический прием, с помощью которого реализовано отделение системного стандартного контекста от стандартного контекста программ пользователей (см. п. 4.6 гл. 1).

что динамически вызываемые процедуры и программы сохраняют свой собственный статус. Например, непривилегированный пользователь может вызывать привилегированные процедуры операционной системы, находящиеся в доступном ему контексте. При этом последние сохраняют право выполнять привилегированные действия. *

1.8. Классификация переменных и констант. Ниже перечислены разновидности переменных и констант с указанием пунктов, где даны подробные определения.

Различаются следующие разновидности переменных: простые переменные (п. 6.1), элементы массивов (п. 6.2), поля переменных (п. 6.3) и формальные параметры (п. 11.1).

В языке введено несколько разновидностей констант: простые константы (§ 5), элементы константных массивов (п. 6.2), параметры циклов (п. 9.4), статические ситуации (п. 10.1), процедуры-константы (п. 11.1), базы участков базированных областей памяти (§ 16), метки предложений (§ 19).

Существуют также препроцессорные константы: описатели полей (п. 6.3) и текстовые макросы (§ 17). Особенность этих констант состоит в том, что они используются только во время трансляции, как макросредства.

1.9. Дополнительные возможности. В язык встроены групповые операции обработки битовых, цифровых и литерных векторов. В число этих операций входит пересылка векторов, сравнение, поиск определенного элемента, операции перевода чисел (двоичное ↔ шестнадцатеричное ↔ литерное). Для векторов, состоящих из элементов простых форматов, введены только пересылка и поиск.

Определение текста и подстановка текста рассматриваются, как простые макросредства, и предназначены для сокращения и структурирования текста программ.

Конструкции, связанные с преобразованием формы массива, позволяют изменить логическую структуру многомерного массива, обрабатывать сечение массива, как самостоятельный объект и пр. При этом элементы исходного массива не копируются в какой-либо другой массив, создаваемый явно или неявно.

1.10. Метод описания синтаксиса языка. Синтаксис языка описан с помощью порождающих правил для понятий. Каждое правило имеет следующий вид

левая часть ::= правая часть

Синтаксические понятия заключаются в угловые скобки. Альтернативы в правой части разделяются вертикальной чертой.

Если в некоторой альтернативе часть конструкции может отсутствовать, то она заключается в фигурные скобки. Три точки после синтактического понятия означают, что опо может повторяться один или более раз. Аналогично три точки после части конструкции, заключенной в фигурные скобки, означают, что она может повторяться нуль или более раз.

Если в правой части порождающего правила некоторое понятие имеет вид ⟨список X⟩, где X — синтаксическое понятие, то тем самым неявно задано правило:

⟨список X⟩::= ⟨X⟩ {, ⟨X⟩}...

Если в описании семантики употребляется синтаксическое понятие, то оно также заключается в угловые скобки.

Чтобы не перегружать синтаксис языка, некоторые уточнения синтаксического характера вынесены в семантику.

2. Базовые обозначения

2.1. Базовые символы. Конструкции языка представляются с помощью набора базовых символов, являющегося подмножеством кода ДКОИ-8. В этот набор входят:

— русские и латинские буквы верхнего регистра

— цифры от 0 до 9

— специальные символы [. < (+ @] ¤ *) & ; : ' # / | > , % ! " — = _^

— пробел.

2.2. Лексические элементы. Лексический элемент языка — это ⟨идентификатор⟩, ⟨число⟩, ⟨поэлементное представление⟩, подчеркнутый идентификатор или разделитель. Лексические элементы строятся из ⟨букв⟩, ⟨цифр⟩, ⟨двоичных цифр⟩, ⟨восьмеричных цифр⟩, ⟨шестнадцатеричных цифр⟩ и ⟨литер⟩.

⟨буква⟩ ::= A | B | C | D | E | F | G | H | I | J | K
| L | M | N | O | P | R | S | T | U | V | W | X | Q
| Y | Z | Ю | Б | Ц | Д | Ф | Г | Й | Л | Я | Ч
| Ж | Ы | Ь | З | Ш | Щ | Э | Ъ | У

⟨цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨двоичная цифра⟩ ::= 0 | 1

⟨восьмеричная цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

⟨шестнадцатеричная цифра⟩ ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| A | B | C | D | E | F |

⟨литера⟩ ::= ⟨буква⟩ | ⟨цифра⟩ | ⟨вертикальная черта⟩
| ⟨кавычка⟩ | [| . | < | (| + | @ | ^ |] | ¤ | *
|) | ; | : | ' | # | / | , | % | > | ! | -|&
| = | — | ⟨пробел⟩

В тексте книги ⟨буквы⟩ языка набраны курсивом. Следует учитывать, что буквы И и U, П и N имеют разные начертания в верхнем регистре, но одинаковые в нижнем. В дальнейшем, где не оговорено особо, подразумевается, что *и* и *п* представляют соответствующие буквы русского алфавита.

⟨Вертикальная черта⟩ — это символ « | ». ⟨Кавычка⟩, встречающаяся внутри ⟨поэлементного представления⟩ (т. е. не являющаяся ограничителем), обозначается двумя кавычками (""). —

Числовая величина ⟨литеры⟩, задается ее внутренним двоичным представлением (см. Приложение 1). В операциях сравнения считается, что литеры упорядочены по их числовым величинам.

Идентификаторы используются для обозначения переменных и констант, вводимых с помощью описаний.

⟨идентификатор⟩ ::= ⟨буква⟩{⟨буква или цифра⟩}...

⟨буква или цифра⟩ ::= ⟨буква⟩ | ⟨цифра⟩

Числа ⟨Целое⟩ и ⟨вещественное⟩ — это десятичные числа в их обычном понимании. ⟨Порядок⟩ представляет собой степень числа 10.

⟨целое⟩ ::= ⟨цифра⟩...

⟨вещественное⟩ ::= ⟨целое⟩.⟨целое⟩{⟨порядок⟩} | ⟨целое⟩⟨порядок⟩

⟨порядок⟩ ::= *e*{⟨знак⟩}⟨целое⟩

⟨знак⟩ ::= + | —

Поэлементное представление используется для обозначения строк, наборов и для обозначения чисел во внутреннем представлении.

⟨поэлементное представление⟩ ::=

1''⟨двоичная цифра⟩...'' | 3''⟨восьмеричная цифра⟩...''
| 4''⟨шестнадцатеричная цифра⟩...'' | 8''⟨литера⟩...''

⟨Поэлементное представление⟩ задает смежную последовательность двоичных, восьмеричных, шестнадцатеричных и байтовых элементов. Цифра, предшествующая первой кавычке, определяет размер элемента в разрядах и способ кодировки. Длина последовательности (в разрядах) равна произведению размера элемента на количество элементов.

Подчеркнутые идентификаторы используются для представления служебных слов. Подчеркнутый идентифика-

тор — это последовательность, состоящая из символа подчеркивания «_», за которым следует ⟨идентификатор⟩. В описании языка подчеркнутые идентификаторы выделены полужирным шрифтом.

⟨формат⟩ ::= ⟨простой формат⟩ | ⟨строчный формат⟩
⟨простой формат⟩ ::= ф32 | ф64 | ф128
⟨строчный формат⟩ ::= ф1 | ф4 | ф8

Разделители — это специальные символы и следующие слитные сочетания:

: := <:= <=> ** :=: +: −: /: *: ^:
>= <=> <> >/ </ >=/ <=/ =/ <>/
// && %*

2.3. Структура текста программы. Чтобы транслятор мог обработать текст программы, последний представляют в виде файла на каком-либо внешнем носителе, например на перфокартах. Файл текста программы имеет такую же структуру, как и любой другой текстовый файл (подробно см. § 12 гл. 2). Содержимое файла текста программы строится из строк текста *). Строка представляет собой последовательность базовых символов.

2.4. Комментарии и прагматы. Комментарий может быть коротким и длинным. Короткий комментарий начинается символом % и заканчивается концом той строки, на которой он начался. Непосредственно за символом % не может следовать символ *. Длинный комментарий — это последовательность символов, которая начинается и заканчивается ограничителем длинного комментария. Таким ограничителем является пара символов %*. Длинный комментарий может занимать несколько строк.

Прагматом называется последовательность символов, начинающаяся символом ¤ и заканчивающаяся концом той строки, на которой она началась.

Комментарии полностью игнорируются транслятором и не влияют на выполнение программы. В отличие от этого прагматы могут рассматриваться транслятором, редактором и другими компонентами систем программирования как руководство по режиму трансляции, редактирования и пр.

2.5. Использование пробелов. Пробел может встречаться везде в тексте программы, кроме как внутри лексических элементов. Исключение составляет использование пробела в качестве

---

*) Далее в пп. 2.3—2.5 вместо термина «строка текста программы» будем говорить просто «строка».

⟨литеры⟩ в ⟨поэлементном представлении⟩. Переход с одной строки на другую эквивалентен пробелу, кроме тех случаев, когда он заканчивает короткий комментарий или прагмат. Таким образом, лексический элемент должен располагаться на одной строке. Комментарий может встречаться везде, где разрешен пробел.

Последовательность из двух лексических элементов, первый из которых заканчивается ⟨буквой⟩ или ⟨цифрой⟩, а второй начинается с ⟨буквы⟩ или ⟨цифры⟩, должна быть разделена по крайней мере одним пробелом. В частности, пробелом должны быть разделены подчеркнутый идентификатор и следующие за ним ⟨идентификатор⟩ или ⟨поэлементное представление⟩.

3. Изображения

⟨Изображения⟩ явно задают объекты и их информационное содержимое. Они служат для обозначения чисел, наборов, константных массивов, процедур, константных файлов, константных справочников и пустых объектов.

⟨изображение⟩ ::=
⟨число⟩ | ⟨набор⟩
| ⟨константный вектор⟩ | ⟨текст процедуры⟩
| ⟨константный файл⟩ | ⟨константный справочник⟩
| ⟨пустой объект⟩

В этом разделе описываются изображения чисел, наборов и пустых объектов. ⟨Константный массив⟩ рассматривается в п. 6.2, процедура — в п. 11.1, ⟨константный файл⟩ — в § 13 гл. 3, а ⟨константный справочник⟩ — в § 14 гл. 3.

3.1. Изображение числа.

⟨число⟩::=
{ ⟨тег целого⟩} ⟨целое⟩
| { ⟨тег вещественного⟩} ⟨вещественное⟩
| ⟨тег вещественного⟩ ⟨целое⟩
| ⟨тег числа⟩ ⟨поэлементное представление⟩
| ⟨тег⟩

⟨тег целого⟩ ::= **цел32** | **цел64**

⟨тег вещественного⟩ ::= **вещ32** | **вещ64** | **вещ128**

⟨тег числа⟩ ::= ⟨тег целого⟩ | ⟨тег вещественного⟩

Формат числа определяется ⟨тегом числа⟩, указанным в начале изображения. Если тег опущен, то выбирается формат ф64. Количество значащих десятичных цифр в мантиссе (κ) и

диапазон представления абсолютной величины (x) чисел имеют следующие границы *):

целое ф32

$\kappa \leqslant 10$

$0 \leqslant x \leqslant 2147483647$

целое ф64

$\kappa \leqslant 19$

$0 \leqslant x \leqslant 9223372036854775807$

вещественное ф32

$\kappa \leqslant 8$

$8.636168e - 78 \leqslant x \leqslant 7.2370056e75$

вещественное ф64

$\kappa \leqslant 17$

$8.6361685550944446e - 78 \leqslant x \leqslant 7.2370055773322621e75$

вещественное ф128

$\kappa \leqslant 34$

$7.9853553994691991450468625722099 30e - 19728 \leqslant$

$x \leqslant 1.2522062065042790406119202197251 57e19727$

* Внутреннее представление вещественных чисел имеет нормализованный вид. Вещественное с максимальным по абсолютной величине отрицательным порядком задает вещественный нуль. Форма:

⟨тег вещественного⟩ ⟨целое⟩

обозначает вещественное число, ближайшее к изображенному ⟨целому⟩.

Для обозначения чисел во внутреннем представлении используется форма

⟨тег числа⟩ ⟨поэлементное представление⟩

⟨Поэлементное представление⟩ задает ВРП числа, ⟨тег числа⟩ — тип и формат числа. Можно использовать двоичное, восьмеричное, шестнадцатеричное, литерное или смешанное представление с различным форматом элементов. В последнем случае изображенные последовательности независимо от формата элементов плотно сцепляются. Результирующий набор битов прижимается вправо, и, если необходимо, дополняется слева нулями. Чтобы в подобной форме изобразить ВРП числа формата ф128, необходимо указать все 128 разрядов этого значения.

*) Представленные здесь числа выведены форматной печатью на МВК Эльбрус-1.

Как видно из правила для изображения чисел, ⟨тег⟩ может быть использован самостоятельно. В этом случае он рассматривается как средство для обозначения целых чисел формата ф64, поставленных в соответствие типам. Например, типу указатель соответствует ⟨тег⟩ **деск**, обозначающий число 0.

⟨тег⟩ : = ⟨тег числа⟩ | **наб** | **мпроц** | **деск** | **мпход**
| **п32** | **п64**

Соответствие тегов и типов, а также числовая величина тегов приведены в Приложении 2. *

3.2. Изображение набора. Содержимое значащей части набора задается с помощью ⟨поэлементного представления⟩. Длина значащей части равна длине ⟨поэлементного представления⟩ и не должна превышать 64 разрядов.

⟨набор⟩ ::= ⟨поэлементное представление⟩ ... | **истина**
| **ложь** | ""

Результирующий набор — это объект формата ф64, в котором значащая часть прижата вправо. В зависимости от операции один и тот же набор можно рассматривать как битовый, цифровой, литерный или как целое без знака (см. п. 1.4).

Логические значения обозначаются как **истина** и **ложь**. Логическое значение представляется одноэлементным битовым набором. Значение элемента равно либо 1 (**истина**), либо 0 (**ложь**). Две слитные кавычки "" обозначают пустой набор, т. е. набор нулевой длины. При изображении литерного набора разрешается опускать цифру 8, предшествующую первой открывающей кавычке.

* Допускается смешанное представление набора с помощью нескольких ⟨поэлементных представлений⟩ с различным форматом элементов. В этом случае изображенные последовательности независимо от формата элементов плотно сцепляются, прижимаются вправо и образуют значащую часть набора.

3.3. Пустой объект. Изображение пустого объекта используется для обозначения неинициализированных данных:

⟨пустой объект⟩ ::= **пусто32** | **пусто64**

где **пусто32** обозначает пустой объект формата ф32, а **пусто64** — пустой объект формата ф64. *

4. Описания

⟨Описание⟩ вводит переменную или константу и связывает с ней идентификатор. Диапазон текста программы, внутри которого эта связь существует, называется областью действия описания.

В программе может быть несколько описаний с одним и тем же идентификатором. За счет текстуальной вложенности

конструкций области действия подобных описаний могут перекрываться. Тогда идентификатор в данной точке текста программы понимается в том смысле, который ему придан в минимальной охватывающей эту точку области действия.

Контекстом идентификаторов для данной точки текста программы называется множество идентификаторов, описание которых действует в этой точке.

С помощью ⟨описаний⟩ вводятся простые переменные (п. 6.1) и следующие константы: простые константы (§ 5), статические ситуации (предварительное описание, п. 10.1), процедуры-константы (п. 11.1), базы участков базированных областей памяти (§ 16), метки предложений (§ 19), описатели полей (п. 6.3) и текстовые макросы (§ 17).

⟨Описание⟩ может встречаться в составе последовательности описаний в начале ⟨закрытого предложения⟩ или ⟨последовательного предложения⟩. Кроме того, существуют специфические описания, являющиеся составной частью связанных с ними конструкций. К ним относятся описание формальных в заголовке процедуры (п. 11.1), описания статических ситуаций в заголовке структурного предложения (п. 10.1) и описание параметра цикла в заголовке цикла (п. 9.4). Область действия описаний указывается в последующих разделах при изложении семантики этих конструкций. Общие для всех случаев ограничения состоят в следующем:

— внутри одной и той же цепочки описаний один и тот же идентификатор не может быть описан дважды;

— описание идентификатора должно предшествовать его использованию.

⟨описание⟩::=

⟨описание простых переменных⟩ | ⟨описание базы⟩
| ⟨описание простых констант⟩ | ⟨описание меток⟩
| ⟨описание процедур-констант⟩ | ⟨описание полей⟩
| ⟨описание статических ситуаций⟩ | ⟨описание текстов⟩

5. Простые константы

⟨Описание простых констант⟩ вводит константу, значением которой является значение ⟨выражения⟩, находящегося в правой части ⟨идентификации константы⟩. Одно описание может вводить несколько констант.

⟨описание простых констант⟩ ::=

конст ⟨список идентификации констант⟩

⟨идентификация константы⟩ ::=

⟨идентификатор⟩ = ⟨выражение⟩

Если в правой части ⟨идентификация константы⟩ находится выражение статического класса (см. п. 7.10), то вновь созданная константа является константой статического класса.

* ⟨Выражение⟩ из правой части ⟨идентификации константы⟩ выполняется в момент выполнения описания. Для констант статического класса память не отводится. В отличие от этого для констант динамического класса отводится область памяти нужного формата. При необходимости, используя ⟨формирование указателя⟩, можно создать указатель, описывающий константу динамического класса. С помощью этого указателя можно работать с константой косвенными средствами; попытка изменить значение подобной константы приводит к возникновению ситуации *нарушение защиты*.

В правой части ⟨идентификации константы⟩ нельзя рекурсивно использовать ⟨идентификатор⟩ левой части. *

6. Переменные и массивы

Переменная простого формата может быть введена с помощью ⟨описания простых переменных⟩ или ⟨описания формальных⟩ параметров процедуры, например:

```
ф64 x , y , z , p   % четыре переменные формата "слово"
ф128 xx             % переменная формата "два слова"
```

Простая переменная обозначается с помощью идентификатора, введенного в ее описании.

Массив. В языке не существует статического описания объекта типа массив. Такой объект создается динамически с помощью ⟨генератора массива⟩ или явно задается с помощью изображения. Обе конструкции выдают описатель массива (указатель с-вектора или указатель паспорта в-массива). Например, в результате присваивания:

```
x := лок вект [100] ф64
```

значением переменной x станет указатель с-вектора, состоящего из 100 элементов формата ф64.

Компонента массива обозначается с помощью конструкции ⟨элемент массива⟩, которая имеет вид:

⟨первичное⟩[{⟨формат⟩}⟨список выражений⟩]

⟨Первичное⟩ задает массив, а ⟨выражения⟩ определяют индексы нужной компоненты. Например, x[99] обозначает последний элемент вектора x (нумерация элементов начинается от 0).

Подмассив — это подмножество элементов массива. В случае с-вектора указатель, описывающий нужную часть вектора (подвектор), формируется с помощью конструкции

⟨подмассив⟩, которая имеет вид:

⟨первичное⟩[{⟨формат⟩}⟨диапазон⟩]

где ⟨первичное⟩ задает исходный массив, а диапазон определяет подмножество. Например, *x*[10:20] формирует указатель, описывающий часть исходного вектора от элемента с номером 10 до элемента с номером 29, т. е. 20 элементов. Аналогичные средства существуют и для в-массива (см. § 15). Подмассив является массивом в том смысле, что его описатель можно использовать в тех же конструкциях, что и описатель массива.

⟨Формат⟩ позволяет задавать нужную часть с-вектора в терминах элементов, формат которых отличен от исходного. Например, конструкция *x*[ф8 80:160] формирует указатель, описывающий ту же часть вектора, что и *x*[10:20], но в терминах литерного формата. Аналогичный смысл придается компоненте ⟨формат⟩ в конструкции ⟨элемент массива⟩.

Нестандартный формат. Любую часть переменной, как и любую часть с-вектора, можно трактовать как переменную нестандартного формата. В первом случае подобная переменная называется полем переменной, а во втором — полем вектора. Размер поля не превышает 64 разрядов. Поля используются для плотной упаковки информации.

Поле переменной обозначается с помощью конструкции, имеющей вид:

⟨переменная⟩.⟨описатель поля⟩

Первая компонента обозначает переменную, а ⟨описатель поля⟩ задает некоторую ее часть, например *z*.[20:10] обозначает поле переменной *z*, начинающееся с 20-го разряда (нумерация справа налево от 0) и имеющее длину 10 разрядов.

* Поле вектора, как и подмассив, описывается указателем. Последний формируется с помощью конструкции ⟨подмассив⟩. Для косвенного доступа к полю нужно применить конструкцию ⟨указуемая переменная⟩ (см. ниже). Например, после выполнения присваивания:

y := лок вект [100] ф1

конструкция *y*[50:20] @ будет обозначать поле вектора, начинающееся с 50-го элемента (нумерация элементов в массивах слева направо) и имеющее длину 20 элементов. Поле вектора может физически начинаться в одном слове памяти и заканчиваться в другом.

Косвенный доступ к переменным. В языке существует возможность сформировать указатель, описывающий область памяти, занимаемую переменной. Используя это свойство, можно организовать косвенный доступ к переменной. Для этого сначала надо образовать указатель переменной. Это осуществляется

с помощью одноместной операции формирования указателя, которая обозначается символом @. Например, в результате присваивания

$p := @x$

значением переменной p становится указатель переменной x. Для косвенного обозначения переменной используется конструкция ⟨указуемая переменная⟩, которая имеет вид:

⟨первичное⟩ @

Значением ⟨первичного⟩ является указатель переменной. Например, с учетом рассмотренного выше присваивания, p @ обозначает переменную x, а не p. В частности, предложение p @ := 3.14 присваивает переменной x новое значение 3,14. Аналогично (используя формирование указателя) можно образовать указатель для элемента массива и поля переменной. В отличие от этого указатель, описывающий поле вектора, формируется с помощью конструкции ⟨подмассив⟩. Последующее преобразование подвектора в переменную осуществляется конструкцией ⟨указуемая переменная⟩, как показано выше.

Преобразование переменной в вектор. Другое использование указателя переменной основано на том, что указатель позволяет интерпретировать переменную как с-вектор, состоящий — в зависимости от формата переменной — из одного или нескольких элементов. В связи с этим к указателю переменной применимы те же средства, что и к указателю вектора. Например, конструкция ⟨подмассив⟩ позволяет выполнить действие, которое можно трактовать, как преобразование переменной простого формата в вектор элементов строчного формата:

p [ф8 0 : 8]

В результате формируется указатель с-вектора, состоящего из 8 элементов байтового формата. ■

Значение переменной — это значение, полученное переменной в результате последнего выполненного присваивания. Значение (простой объект) хранится в переменной вместе с его типом. В связи с этим одна и та же переменная может принимать значения различного типа. Однако формат переменной накладывает ограничения на возможный формат (а следовательно и тип) значения:

1. Переменная простого формата: формат значения меньше либо равен формату переменной.

2. Переменная строчного формата, поле переменной и поле вектора используются для хранения информации в упакованном виде (см. п. 1.4).

6.1. Простые переменные. ⟨Описание простых переменных⟩ вводит одну или несколько переменных одинакового ⟨простого формата⟩ и связывает с ними ⟨идентификаторы⟩.

⟨описание простых переменных⟩ ::=
⟨простой формат⟩ ⟨список идентификации переменных⟩
⟨идентификация переменной⟩ ::=
⟨идентификатор⟩ { := ⟨выражение⟩}
⟨идентификатор⟩=⟨идентификатор⟩

⟨Идентификация переменной⟩ может содержать ⟨выражение⟩, значение которого задает начальное значение переменной. В этом ⟨выражении⟩ нельзя использовать ⟨идентификатор⟩ из левой части ⟨идентификации переменной⟩.

* По умолчанию переменная формата ф32 или ф64 инициализируется пустым объектом соответствующего формата, а переменная формата ф128 — пустым объектом формата ф64 *).
⟨Идентификация переменной⟩ выполняется в следующем порядке. Сначала вычисляется значение ⟨выражения⟩, затем для переменной отводится область памяти заданного формата и осуществляется запись начального значения.
Вторая альтернатива правила для ⟨идентификация переменной⟩ предназначена для описания переменной, эквивалентной по адресу некоторой другой ранее описанной переменной. Последняя задается в правой части. Формат новой переменной может быть меньше, равен или больше формата эквивалентной переменной. В последнем случае новая переменная накладывается также и на те, которые следуют за эквивалентной переменной. *

В области действия описания переменная обозначается ⟨идентификатором⟩, связанным с нею при описании.

6.2. Массив. Массив создается с помощью ⟨генератора массива⟩. ⟨Генератор в-массива⟩ предназначен для создания многомерного в-массива, а ⟨генератор с-вектора⟩ — для создания с-вектора.

⟨генератор массива⟩ ::=
⟨генератор в-массива⟩ | ⟨генератор с-вектора⟩
⟨генератор в-массива⟩ ::=
⟨локализация⟩[⟨список выражений⟩]⟨формат⟩
⟨генератор с-вектора⟩ ::=
⟨локализация⟩**вект**[⟨выражение⟩{:⟨выражение⟩}]⟨формат⟩
⟨локализация⟩ ::= **лок** | **глоб**

В-массив состоит из элементов заданного ⟨формата⟩. Размерность массива равна числу ⟨выражений⟩ в скобках. Индекс по p-му измерению может заключаться в пределах от 0 до $k_p - 1$, где k_p — значение p-го ⟨выражения⟩ (k_p — целое неотрицательное число).

---

*) Точнее, в каждое из двух слов, отведенных для переменной, помещается пустой объект формата ф64.

Значением генератора является указатель паспорта созданного массива. Например, в результате выполнения генератора

лок [10,10] **ф64**

создается матрица размером 10×10 элементов формата ф64. Если генератор массива использовать в правой части описания константы, например

конст *с* = **лок** [10 , 10] **ф64** ,

то образуется описание, эквивалентное традиционному описанию массива.

С-вектор состоит из *к* элементов заданного формата, где *к* — значение первого ⟨выражения⟩. Значением генератора является указатель, описывающий созданный вектор.

Локализация. Компонента ⟨локализация⟩ определяет время жизни созданного массива: **лок** означает, что массив является локальным, а **глоб** — глобальным.

* Попытка доступа к уничтоженному массиву (локальному и глобальному) приводит к возникновению ситуации *ситнетмас*.

Уничтожение массива можно осуществить явно с помощью стандартной процедуры *откреп*, имеющей один параметр:

откреп (*а*)

где *а* — указатель с-вектора или указатель паспорта в-массива.

Вектор плавающей длины. При выполнении конструкции ⟨генератор с-вектора⟩ с двумя ⟨выражениями⟩ создается вектор плавающей длины. Значение первого ⟨выражения⟩ задает начальное количество элементов, а значение второго ⟨выражения⟩ — максимально возможное число элементов. Изменение длины вектора осуществляется с помощью стандартной функции *измдлину*. Эта функция имеет два параметра:

измдлину (*а*, *к*)

где значение параметра *а* — указатель вектора плавающей длины, а значение *к* — целое число, задающее приращение длины вектора (при положительном *к*) или уменьшение его длины (при отрицательном *к*). Значением функции является указатель, описывающий результирующий вектор.

Первый параметр можно передавать не только значением, но также именем

измдлину (**имя** *а*, *к*)

где *а* — обозначает переменную (или константу динамического класса), значением которой является указатель вектора. После выполнения функции новым значением переменной становится указатель результирующего вектора. Это позволяет использовать вызов этой функции в качестве ⟨оператора⟩. *

Изображение вектора. Конструкция ⟨константный вектор⟩ предназначена для непосредственного изображения с-векто-

ров в тексте программы. Значение элементов константного вектора не изменяется в процессе выполнения программы.

* Попытка изменить значение приводит к возникновению ситуации *нарушение защиты*. *

⟨константный вектор⟩ ::= ⟨массив констант⟩ | ⟨строка⟩

⟨Массив констант⟩ — это изображение с-вектора, сформированного из констант, перечисленных в круглых скобках.

⟨массив констант⟩ ::=
мконст⟨формат⟩(⟨список элементов массива констант⟩)
⟨элемент массива констант⟩ ::=
{⟨число повторений⟩}⟨выражение⟩
| {⟨число повторений⟩}(⟨список элементов массива констант⟩)
⟨число повторений⟩ ::= /⟨целое⟩/

⟨Выражение⟩, задающее значение элемента, может быть выражением статического класса либо ⟨строкой⟩. Во втором случае подразумевается, что в состав массива входит содержимое строки (а не указатель, описывающий строку).

Значением конструкции является указатель, описывающий вектор в терминах заданного ⟨формата⟩.

* Элементы массива в общем случае могут иметь различные форматы, возможно, не совпадающие с заданным ⟨форматом⟩. Массив заполняется в соответствии с форматом значений элементов. Практическим следствием такой упаковки является следующее: последовательность элементов формата ф32 располагается смежно в нужном количестве слов; строки всегда начинаются с нового слова; вложенные упакованные списки элементов начинаются с нового слова. *

⟨Строка⟩ — это изображение битового, цифрового или литерного константного с-вектора, состоящего из элементов, заданных между кавычками. Элементы пумеруются слева направо, начиная с нуля. Значением конструкции является указатель, описывающий вектор в терминах формата, заданного служебным словом: **стр1** (ф1), **стр4** (ф4), **стр8**(ф8).

⟨строка⟩::=
стр1 ⟨поэлементное представление⟩...
| **стр4** ⟨поэлементное представление⟩...
| **стр8** ⟨поэлементное представление⟩...

* ⟨Поэлементные представления⟩ сцепляются таким же образом, как и в случае наборов. Допускается смешанное представление строки. Кодировку формата в первом ⟨поэлементном представлении⟩ можно опустить в случае, если она совпадает

с результирующим форматом. Например, строка стр8 8"литеры" эквивалентна строке стр8 "литеры". Путем разбиения ⟨строки⟩ на ⟨поэлементные представления⟩ ее можно переносить с одной строки текста программы на другую. *

Элемент массива. Конструкция, используемая для обозначения элемента массива, имеет вид:

⟨элемент массива⟩ ::=
⟨первичное⟩ [{⟨формат⟩} ⟨список выражений⟩]

С помощью ⟨элемента массива⟩ может обозначаться:

1. Элемент выстроенного массива. В этом случае значение ⟨первичного⟩ приводит к в-массиву.

2. Элемент смежного вектора. Значение ⟨первичного⟩ приводит к с-вектору.

В обоих случаях индексы элемента определяются значениями ⟨выражений⟩. Количество ⟨выражений⟩ должно быть равно числу измерений.

* В случае в-вектора форма ⟨первичного⟩ ограничена. Она может быть представлена: (а) конструкцией, создающей паспорт в-вектора, (б) идентификатором константы, в правой части описания которой находится одна из таких конструкций, (в) подстановкой текста, в правой части описания которого находится одна из таких конструкций. Упомянутыми конструкциями являются: генератор одномерного в-массива, ⟨формирование паспорта⟩ одномерного в-массива, вызов функции *ввект*, генератор паспорта одномерного в-массива. Последние три конструкции описаны в § 15. Два основных случая демонстрируются примерами 1 и 2.

Пример 1. Случай константы.

```
начало
    конст в = лок [n] ф64;
    ... в [i] ...
конец
```

Пример 2. Случай параметра-массива.

```
проц (в)
начало
    текст вв = ввект (в);
    ... вв [i] ...
конец
```

Во втором примере предполагается, что фактическим параметром процедуры является паспорт в-вектора.

Преобразование формата. При обращении к элементу с-вектора можно указать ⟨формат⟩, возможно, не совпадающий с тем, который предписан указателем. В этом случае осуществляется индексация в терминах указанного ⟨формата⟩. По умолчанию формат элемента полагается равным формату, предписанному указателем.

Диапазон индексов. Индекс по каждому измерению является целым числом и должен заключаться в диапазоне,

разрешенном для данного измерения. Массив с нижними границами, отличными от нуля, может быть получен только в результате использования конструкции ⟨формирование паспорта⟩ (см. § 15). Если индексы таковы, что смещение селектируемого элемента относительно начала массива больше его длины, возникает ситуация *границамассива*. *

Подмассив. Конструкция ⟨подмассив⟩ формирует указатель, описывающий часть с-вектора.

⟨подмассив⟩ ::= ⟨первичное⟩[{⟨формат⟩}⟨диапазон⟩]
⟨диапазон⟩ ::= {⟨выражение⟩} : {⟨выражение⟩}

Выделяемая часть вектора определяется значениями ⟨выражений⟩. Значение первого ⟨выражения⟩ задает номер начального элемента. Значение второго ⟨выражения⟩ задает количество элементов. Сформированный указатель описывает заданную часть, как с-вектор, элементы которого нумеруются от 0 до $к-1$, где $к$ — количество элементов.

В случае, если одно или оба ⟨выражения⟩ опущены, их значения выбираются по умолчанию. В качестве начального индекса диапазона выбирается индекс 0. Длина диапазона по умолчанию полагается равной количеству элементов от начального элемента диапазона до конца вектора.

* Преобразование формата. В конструкции ⟨подмассив⟩ компонента ⟨формат⟩ используется с тем же смыслом, что и в конструкции ⟨элемент массива⟩. В этом случае результирующий указатель описывает подвектор в терминах нового формата. Это свойство можно, в частности, использовать для преобразования формата элемента вектора. Например, если значением $х$ является указатель вектора, состоящего из $к$ элементов словного формата, то $х$[ф8:] выдает указатель, описывающий ту же память, что и вектор из $8*к$ элементов литерного формата.

Если значения ⟨выражений⟩ таковы, что заданная часть выходит за рамки исходного вектора, возникает ситуация *границамассива*. *

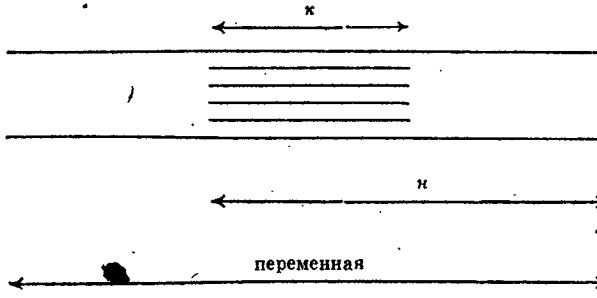
6.3. Описание и использование полей.

В этом разделе описывается способ обозначения поля переменной и семантика выборки поля. Семантика присваивания полю изложена в § 8.

⟨поле переменной⟩ ::= ⟨переменная⟩.⟨описатель поля⟩
⟨описатель поля⟩ ::=
  ⟨непосредственный описатель поля⟩
  |⟨идентификатор⟩
⟨непосредственный описатель поля⟩ ::=
  [{⟨строчный формат⟩}⟨выражение⟩{:{⟨выражение⟩}}]

При определении координат поля считается, что память, занимаемая исходной переменной, состоит из элементов указанного ⟨строчного формата⟩, пронумерованных справа налево, начиная с номера 0. По умолчанию выбирается битовый формат.

Местоположение поля определяется по следующей схеме:



где $н$ — значение первого ⟨выражения⟩, т. е. номер начального элемента поля; $к$ — значение второго ⟨выражения⟩, задающего длину поля; $н - к + 1$ — номер последнего элемента поля.

Местоположение поля либо задается непосредственно, либо указывается с помощью ⟨идентификатора⟩ поля, являющегося синонимом некоторого ⟨непосредственного описателя поля⟩ (см. ниже семантику ⟨описания поля⟩).

Выборка поля. Конструкция ⟨поле значения⟩ осуществляет выборку поля из значения ⟨первичного⟩. В большинстве практических случаев эту и предыдущую конструкцию можно считать синтаксически эквивалентными.

* Отличие ⟨поля значения⟩ состоит лишь в том, что (а) оно может быть только элементом ⟨выражения⟩ и (б) его исходный операнд может быть представлен произвольным выражением. *

⟨поле значения⟩ ::= ⟨первичное⟩.⟨описатель поля⟩

Местоположение поля определяется, как указано выше, но вместо переменной рассматривается ВРП значения ⟨первичного⟩. Результатом выборки является набор, значащая часть которого представляет собой содержимое заданного поля.

* Если в конструкции опущены:

— символ двоеточия и второе ⟨выражение⟩, то считается, что поле состоит из одного элемента;

— второе ⟨выражение⟩, то рассматривается поле от начального элемента до правого края исходного объекта (в случае ⟨поля значения⟩) или до правого края исходной переменной (в случае ⟨поля переменной⟩).

Исходный объект не может иметь удвоенный формат. Значения ⟨выражений⟩ в ⟨описателе поля⟩ должны быть неотрицательными целыми.

В перечисленных ниже случаях семантика поля имеет ряд особенностей, связанных с реализацией.

1. Поле числа формата ф32.
2. Присваивание полю переменной формата ф32.
3. Присваивание полю, в случае, когда правая часть присваивания производит побочный эффект, влияющий на левую часть.
4. Присваивание полю целого отрицательного числа.

В первом случае надо учитывать, что ВРП числа формата ф32 занимает 64 разряда. Например, пусть дана переменная t формата ф32. Тогда выборка содержимого знакового разряда изображается как $t.[63]$.

Остальные особенности рассмотрены в § 8.

Формирование значения. ⟨Формирование⟩ предназначено для одновременной замены содержимого нескольких полей в значении ⟨первичного⟩.

⟨формирование⟩ ::=
⟨первичное⟩.(⟨список элементов формирования⟩)
⟨элемент формирования⟩ ::=
⟨описатель поля⟩ : ⟨выражение⟩ | **тип** : ⟨выражение⟩

⟨Описатель поля⟩ в элементе формирования задает некоторое поле в исходном объекте, куда помещается значение ⟨выражения⟩. При этом последнее обрезается слева до длины поля.

По умолчанию в качестве исходного объекта выбирается целое число 0 формата ф64.

Результатом выполнения конструкции является исходный объект с модифицированным содержимым указанных полей.

Если в левой части ⟨элемента формирования⟩ находится служебное слово **тип**, то такой элемент предписывает изменение типа исходного объекта. В этом случае целое, являющееся значением ⟨выражения⟩, задает тип объекта, полученного в результате формирования. Нумерация типов приведена в Приложении 2.

Замечание. Формирование осуществляет замену полей в объекте, а не в переменной. Однако, используя присваивание, можно модифицированный объект присвоить переменной, например:

$x := x.([10:5]:y, [20:10]:z)$ *

Описание поля вводит идентификатор, являющийся синонимом ⟨описателя поля⟩, находящегося в правой части ⟨идентификации поля⟩.

⟨описание поля⟩ ::= **поле** ⟨список идентификации полей⟩

⟨идентификация поля⟩ ::= ⟨идентификатор⟩=⟨описатель поля⟩

В данном случае ⟨описатель поля⟩ может содержать только ⟨выражения⟩ статического класса (см. п. 7.10).

Одно ⟨описание поля⟩ может вводить несколько идентификаторов поля.

Пример.

```
поле p1 = [10 : 5] , p2 = [20 : 10] ;
    ...
    x . p1 := y ;
    ...
    x . p2 := z ;
```

* **6.4. Указатель переменной.** В этом пункте приводятся две конструкции, связанные с косвенным доступом к переменным.

Создание указателя. Операция @ формирует указатель, описывающий переменную.

⟨формирование указателя⟩ ::= @ ⟨переменная⟩

Результирующий указатель описывает переменную как с-вектор, состоящий (в зависимости от ее формата) из одного или нескольких элементов. В случае переменной простого или строчного формата формируется указатель вектора из одного элемента того же формата. В случае поля переменной или поля вектора формируется указатель вектора из k элементов формата ϕ, где k и ϕ — параметры поля.

Косвенное обращение к переменной. Конструкция ⟨указуемая переменная⟩ обозначает переменную, указатель которой является значением ⟨первичного⟩.

⟨указуемая переменная⟩ ::= ⟨первичное⟩ @ *

6.5. Переменная. Синтаксическое правило для ⟨переменной⟩ объединяет все возможные способы обозначения переменных.

⟨переменная⟩ ::=
⟨идентификатор⟩ | ⟨элемент массива⟩
| ⟨поле переменной⟩ | ⟨указуемая переменная⟩
| ⟨закрытая переменная⟩ | ⟨подстановка текста⟩

Здесь ⟨идентификатор⟩ — это обозначение простой переменной или формального параметра. Смысл конструкций ⟨элемент массива⟩, ⟨поле переменной⟩ и ⟨указуемая переменная⟩ рассмотрен выше.

Закрытое предложение (точнее — ⟨закрытая переменная⟩) также может служить для обозначения переменных. В этом случае предложение строится таким образом, чтобы его выпол-

нение завершалось выполнением какой-либо конструкции, предназначенной для обозначения переменных. Последняя и задает нужную переменную. Например, условное предложение в левой части присваивания:

если q **то** x **иначе** y **все** := 3.14

обозначает в зависимости от значения q либо переменную x, либо переменную y.

7. Выражения

Правило для ⟨выражения⟩ объединяет конструкции, которые в результате выполнения выдают значение. Такими конструкциями являются ⟨формулы⟩, представляющие собой традиционную алгебраическую запись правил вычисления значения, ⟨присваивание⟩ переменной, выдающее значение правой части, и ⟨операции над массивами⟩.

⟨выражение⟩ ::=
⟨формула⟩ | ⟨присваивание⟩ | ⟨операции над массивами⟩

7.1. Формула. Формула задает правило для вычисления значения.

⟨формула⟩ ::=
⟨логическая сумма⟩{**экв**⟨логическая сумма⟩} ...
⟨логическая сумма⟩ ::=
⟨логическое произведение⟩{**или**⟨логическое произведение⟩} ...
⟨логическое произведение⟩ ::= ⟨отношение⟩{**и**⟨отношение⟩} ...
⟨отношение⟩ ::=
⟨сумма⟩{⟨операция отношения⟩⟨сумма⟩} ...
| ⟨сравнение строк⟩
⟨сумма⟩ ::=
⟨произведение⟩ { ⟨операция типа сложения⟩
⟨произведение⟩} ...
⟨произведение⟩ ::=
⟨сомножитель⟩{⟨операция типа умножения⟩
⟨сомножитель⟩} ...
⟨сомножитель⟩ ::=
⟨одноместная формула⟩}⟨операция типа степени⟩
⟨одноместная формула⟩} ...
⟨одноместная формула⟩ ::=
{⟨одноместная операция⟩}⟨первичное⟩
| ⟨формирование указателя⟩
⟨первичное⟩ ::=
⟨изображение⟩ | ⟨идентификатор⟩

| ⟨элемент массива⟩ | ⟨подмассив⟩
| ⟨указуемая переменная⟩ | ⟨вызов⟩
| ⟨генератор⟩ | ⟨закрытое выражение⟩
| ⟨формирование паспорта⟩ | ⟨формирование⟩
| ⟨форматный обмен⟩ | ⟨признак⟩
| ⟨подстановка текста⟩ | ⟨запрос атрибута⟩
| ⟨модификация атрибута⟩ | ⟨внешнее имя⟩
| ⟨двоичный обмен⟩ | ⟨поле значения⟩

Значением ⟨первичного⟩ может быть:

1. Объект, заданный изображением (число, набор, указатель вектора и т. д.).
2. Значение переменной или константы, с которой ⟨идентификатор⟩ связан описанием. Сюда относятся значения простых переменных и формальных параметров, а также значения констант: простых констант, процедур-констант, статических ситуаций, параметров циклов, меток, баз участков БОП.
3. Значение элемента массива.
4. Указатель подвектора.
5. Значение поля.
6. Значение переменной, обозначенной косвенным образом.
7. Результат вызова функции.
8. Указатель, описывающий динамически созданный объект.
9. Результат выполнения ⟨закрытого выражения⟩.
10. Указатель паспорта преобразованного массива.
11. Объект, полученный в результате ⟨формирования⟩.

Остальные случаи разобраны в § 13, п. 14.5 и §§ 17, 18 этой главы и §§ 3, 7 гл. 3.

Последовательность ⟨первичных⟩ в ⟨формуле⟩ выполняется слева направо. Последовательность выполнения операций зависит от соотношения их приоритетов: сначала выполняются более приоритетные операции, а затем менее приоритетные. Последовательность операций одинакового приоритета выполняется слева направо.

Ниже описываются соотношение приоритетов операций, смысл операций, типы операндов и тип результата.

* В процессе выполнения операции может быть обнаружена ошибка в исходных данных или может возникнуть переполнение результата. Если тип операнда не соответствует тому, который предписывается операцией (с учетом описанной ниже балансировки типов числовых операндов), возникает ситуация *неверныйоперанд*. Остальные случаи описаны в семантике соответствующих операций. *

7.2. Приоритет операций. Все операции разбиты по приоритетам на восемь групп. Логические двухместные операции имеют

низший приоритет: **экв** — первый, **или** — второй, **и** — третий. Ниже приведены синтаксические правила для остальных операций. Приоритет этих операций выше, чем у логических. Правила расположены в порядке возрастания приоритетов.

⟨операция отношения⟩ ::=
⟨операция сравнения⟩ | **<=>** | **среди**
⟨операция сравнения⟩ ::= **<>** | **=** | **>** | **>=** | **<** | **<=**
⟨операция типа сложения⟩ ::= **+** | **−** | **+:** | **−:** | **плюс** | **минус**
⟨операция типа умножения⟩ ::= ***** | **/** | **умнд** | ***:** | **/:** | **остат**
⟨операция типа степени⟩ ::= *****
⟨одноместная операция⟩ ::=

целокр	**целобр**	**вещокр**	**вещобр**
ф32окр	**ф32обр**	**ф64окр**	**ф64обр**
цел64окр	**цел64обр**	**в128**	**целзн**
стнаб	**млнаб**	**естьнабор**	**естьпусто**
естьцел	**естьвещ**	**естьф32**	**естьф64**
естьф128	**ие**	**пче**	**перв1**
знак	**длина**	**адрес**	**тип**
дес	**бит**		

Операции ⟨сравнения строк⟩ имеют такой же приоритет, как и ⟨операции отношения⟩.

Пример. Выражение

$$x - y > омега * сум * cos(y + 3 * t) ** 2/7.394 - 8/m[i] - a ** 3$$

эквивалентно выражению

$$(x - y) > (((((омега * сум) * (\cos(y + (3 * t)) ** 2))/7.394)) - (8/m[i])) - (a ** 3)).$$

7.3. Логические операции.

Знак операции	Операция	Тип операндов	Тип результата
и	конъюнкция	битовое	битовое
или	дизъюнкция	битовое	битовое
экв	эквивалентность	битовое	битовое
не	отрицание	битовое	битовое

Операции осуществляются поэлементно. Длина результирующего битового равна длине исходных битовых наборов.

Перед операцией производится выравнивание длин исходных наборов: меньший дополняется слева до большего нулевыми элементами,

7.4. Операции отношения.

Знак операции*)	Операция	Тип операндов	Тип результата
= < >	сравнение	вещественное	логическое
> > =		целое	логическое
< < =		набор	логическое

* Двухсимвольные обозначения имеют следующий смысл: < > — не равно, > = — больше либо равно, < = — меньше либо равно.

В случае наборов операция сравнения осуществляет поэлементное сравнение операндов. Сравнение происходит слева направо.

* *Балансировка операндов.* Операции сравнения и приведенные ниже (см. п. 7.5) арифметические операции осуществляют предварительную балансировку исходных операндов. Балансировка заключается в следующем:

1. Если типы операндов различаются, то один операнд преобразуется к типу другого по схеме: набор → целое → вещественное.
2. Если форматы операндов различаются, то операнд меньшего формата преобразуется к большему формату.
3. Длины исходных наборов выравниваются (меньший дополняется слева до большего нулевыми элементами).
4. Порядки исходных вещественных выравниваются (число с меньшим порядком преобразуется к большему порядку).

Преобразование набора в целое происходит следующим образом. Незначащие элементы набора заполняются нулями, и тип объекта заменяется на тип целое формата ф64. В случае исходного полного набора, старший бит которого равен 1 (что соответствует отрицательному знаку результирующего целого), такое преобразование приводит к возникновению ситуации *неверныйоперанд*. При необходимости, для преобразования такого набора нужно пользоваться специальной операцией **целзн.**

Преобразование целого в нормализованное вещественное происходит с сохранением формата. Если отбрасываемая часть мантиссы операнда отлична от нуля, происходит округление: в младший разряд мантиссы результата заносится 1.

Дополнительные операции сравнения.

Знак операции	Операция	Тип операндов	Тип результата
<=>	проверка совпадения	любой	логическое
x среди *y*	проверка принадлежности	*x* — литера *y* — битовая строка	логическое

Результатом операции проверки совпадения является истина в случае, если у операндов совпадают типы, форматы, величины, и в случае наборов — длины.

Результатом операций проверки принадлежности является результат сравнения $y[x] =$ истина. В общем случае операнд x может быть представлен литерным набором, в котором рассматривается правая крайняя литера, или литерной строкой, в которой рассматривается начальная литера. *

7.5. Арифметические операции.

Знак операции	Операция	Тип операндов	Тип результата
+	сложение	вещественное целое	вещественное целое/вещественное
—	вычитание	вещественное целое	вещественное целое/вещественное
*	умножение	вещественное целое	вещественное целое/вещественное
	деление	вещественное целое	вещественное вещественное
умнд	удвоенное умножение	вещественное целое	вещественное целое/вещественное
x**y	возведение в степень	y — целое x — целое x — вещественное	целое вещественное
/:	частное деления нацело	целое	целое
остат	остаток деления нацело	целое	целое

Операция умнд осуществляет умножение с удвоенной точностью. Формат результата в два раза больше формата операндов, но не больше ф128. В случае умножения двух целых формата ф64 результатом является вещественное формата ф128.

Возведение в положительную целочисленную степень осуществляется путем многократного умножения левого операнда самого на себя. В случае отрицательной степени возвращается величина, обратная результату умножения.

Операции целочисленного деления связаны соотношением:

$$x = (x\ /:\ y) * y + (x \text{ остат } y)$$

Здесь x остат y имеет знак x и абсолютную величину меньшую,

чем y. Знак $x/:y$ определяется так же, как в случае обычного деления.

* В операциях деления /, /:, остат возникает ситуация *делениенануль* в случае, если делитель является целым или вещественным нулем.

Вещественный результат арифметической операции представляется в нормализованном виде. Если отбрасываемая часть мантиссы отлична от нуля, происходит округление: в младший разряд мантиссы результата заносится 1.

В случае переполнения вещественного результата возникает ситуация *переполнвещ* (для форматов ф32 и ф64) или ситуация *переполнвещ*128 (для формата ф128). В случае отрицательного переполнения порядка результата выдается вещественный нуль.

Контроль целых. В операциях +, —, * над целыми операндами возможен вещественный результат. Он возникает в том случае, когда абсолютная величина целочисленного результата превышает максимально допустимую для выбранного формата. В целочисленных операциях +:, —:, *: аналогичные обстоятельства приводят к возникновению ситуации *переполнцел32* (для формата ф32) или *переполнцел64* (для формата ф64).

Знак операции	Операция	Тип операндов	Тип результата
+:	сложение целых	целое	целое
—:	вычитание целых	целое	целое
*:	умножение целых	целое	целое

Контроль точности вычисления. Два вещественных числа, имеющие одинаковые мантиссы m и порядки p рассматриваются как близкие по величине числа, различающиеся на погрешность представления, т. е. на величину порядка $(p - кт)$, где $кт$ — максимальное количество шестнадцатеричных цифр в мантиссе вещественного данного формата. Вычитание этих чисел, произведенное операциями — или +, дает не вещественный нуль, а погрешность представления (вещественное число с нулевой мантиссой и порядком, равным $p - кт$). Сложение (или вычитание) погрешности с числом, которое меньше ее по абсолютной величине, а также сравнение погрешности с таким числом означает, что вычисление происходит в области величин, сравнимых с погрешностью представления. При этих обстоятельствах в операциях +, —, а также в операциях сравнения возникает ситуация *потерязначимости*. В остальных операциях погрешность рассматривается как вещественный нуль. Дополнительно введены операции сложения и вычитания без контроля точности:

Знак операции	Операция	Тип операндов	Тип результата
плюс	сложение	вещественное целое	вещественное целое
минус	вычитание	вещественное целое	вещественное вещественное

В этом случае результатом вычитания двух одинаковых вещественных чисел является вещественный нуль. *

7.6. Операции преобразования типа и формата. Одноместные операции преобразования типа и формата определены для числовых типов *).

Знак операции	Операция	Тип операнда	Тип результата
целокр	в целое: с округлением	число	целое
целобр	с обрубанием	число	целое
вещокр	в вещественное: с округлением	число	вещественное
вещобр	с обрубанием	число	вещественное
ф32окр	в формат ф32: с округлением	число	число ф32
ф32обр	с обрубанием	число	число ф32
ф64окр	в формат ф64: с округлением	число	число ф64
ф64обр	с обрубанием	число	число ф64
в128	в формат ф128	число	вещественное ф128
цел64окр	в целое ф64: с округлением	число	целое ф64
цел64обр	с обрубанием	число	целое ф64
целзн	в целое со знаком	набор	целое ф64
стнаб	удвоенное в набор	вещественное ф128	набор
млнаб	удвоенное в набор	вещественное ф128	набор

Результат операций преобразования типа — **целокр, целобр, вещокр, вещобр** сохраняет формат операнда. Результат операций преобразования формата **ф32окр, ф32обр, ф64окр, ф64обр,**

*) Общий случай преобразования типа предусмотрен в конструкции ⟨формирование⟩.

в128 сохраняет тип операнда. Операции **цел64окр, цел64обр,** преобразуют операнд в целое формата ф64.

Преобразование в целое с обрубанием производится путем отбрасывания дробной части. Преобразование в целое с округлением осуществляется по следующей формуле:

$$окр(x) = sign(x) * обр(abs(x) + 0.5),$$

где $sign(x) = 1$ для $x > 0$, 0 для $x = 0$, -1 для $x < 0$,
$обр(x)$ — функция преобразования в целое с обрубанием.

Таким образом, округление — это преобразование вещественного к ближайшему целому.

Преобразование целого в вещественное того же формата и сокращение формата вещественного могут привести к отбрасыванию части мантиссы операнда. В связи с этим есть два варианта соответствующих операций — с округлением и с обрубанием. Округление состоит в том, что в младший разряд мантиссы результата заносится 1 в случае, если отбрасываемая часть отлична от нуля. Обрубание представляет собой отбрасывание без округления.

* Операция **целзн** преобразует полный или неполный битовый набор в целое со знаком формата ф64. Старший бит значащей части набора рассматривается как знак числа и переносится в знаковый разряд.

Преобразование в целое может привести к переполнению результата. В этих случаях возникает ситуация *переполнцел32* (для формата ф32) или ситуация *переполнцел64* (для формата ф64). Преобразование в вещественное может привести к переполнению порядка. Этим случаям соответствует ситуация *переполнвещ* (для форматов ф32 и ф64).

Операции **стнаб** и **млнаб** преобразуют соответственно старшую и младшую половину ВРП вещественного формата ф128 в полный битовый набор. *

7.7. Операции проверки типа и формата. Описываемые ниже операции являются одноместными. Каждая операция проверя-

Знак операции	Операнд	Тип операнда	Тип результата
	проверка на:		
естьцел	тип целое,	любой	логическое
естьвещ	тип вещественное	любой	логическое
естьпусто	тип пусто	любой	логическое
естьнабор	тип набор	любой	логическое
	проверка:		
естьф32	формата ф32	любой	логическое
естьф64	формата ф64	любой	логическое
естьф128	формата ф128	любой	логическое

ет, совпадает ли тип (или формат) операнда с тем типом (или форматом), который в ней указан. В случае совпадения выдается **истина**, иначе — **ложь**.

* **7.8. Операции перевода чисел.** Операция **дес** переводит целое число в десятичное представление. Результат представлен в виде цифрового набора, в котором k-й элемент равен десятичной цифре, находящейся в k-й позиции. Длина набора равна количеству значащих цифр в переведенном числе. Если количество цифр превышает 16, то признак переполнения тгп устанавливается в 1, иначе — в 0. Если исходное число имеет отрицательный знак, признак отношения тго устанавливается в 1 (**истина**), иначе — в 0 (**ложь**).

Операция **бит** осуществляет обратный перевод из десятичного представления в двоичное. Результат представляется в виде целого формата ф64 со знаком (в зависимости от исходного состояния признака отношения результату приписывается положительный или отрицательный знак).

Относительно признака отношения и переполнения см. дополнительно п. 14.5. *

7.9. Прочие одноместные операции.

Знак операции	Операция	Тип операнда	Тип результата
—	изменение знака	вещественное	вещественное
		целое	целое
знак	знак числа	число	целое ф32
длина	длина вектора	с-вектор	целое ф64
пче	число единиц	любой	целое ф32
перв1	первая единица	любой	целое ф32
тип	тип-формат	любой	битовое
адрес	адрес вектора	с-вектор	целое ф64

Результат операции **знак** определяется следующим соотношением:

$$\textbf{знак}\ x = \begin{cases} 1, & \text{если } x > 0, \\ -1, & \text{если } x < 0, \\ 0, & \text{если } x = 0. \end{cases}$$

Операции **длина** и **адрес** выдают соответственно длину и адрес начала с-вектора.

* Операция **пче** определяет число единиц в ВРП операнда, а операция **перв1** — номер старшей единицы (если единиц нет, то результат равен —1). Формат операнда не должен превышать ф64.

Операция **тип** выделяет тип и формат операнда. В поле [7 : 8] результата содержится код типа-формата операнда. Если операнд является набором, то в поле [15 : 8] содержится ко-

личество битовых элементов в наборе. Кодировка типов приведена в Приложении 2. *

7.10. Выражения статического класса. Выражение статического класса — это ⟨формула⟩, в которой ⟨первичными⟩ составляющими могут являться:

а) ⟨число⟩, ⟨набор⟩, ⟨пустой объект⟩;

б) ⟨идентификатор⟩ константы статического класса, т. е. константы, в правой части описания которой находится выражение статического класса;

в) конструкция ⟨поле значения⟩, все компоненты которой являются выражениями статического класса;

г) выражение статического класса в круглых скобках;

д) конструкция ⟨формирование⟩, все компоненты которой являются выражениями статического класса *).

Пример. Выражения статического класса в описаниях констант.

```
конст пи = 3.141593,
      ерадрад = пи/180,
      е = 2.718282,
      еке = е ** ;
```

Выражения статического класса вычисляются во время трансляции.

7.11. Генератор. ⟨Генератор⟩ предназначен для динамического создания составных объектов. Значением ⟨генератора⟩ является указатель соответствующего типа. С помощью этого указателя осуществляется последующий доступ к созданному объекту.

⟨генератор⟩ ::=
⟨генератор массива⟩ | ⟨генератор ситуации⟩
| ⟨генератор объекта связи⟩ | ⟨генератор внешнего объекта⟩
| ⟨генератор паспорта⟩

* *Задержанная генерация.* В общем случае компоненты ⟨генератора⟩ выполняются в текстуальной последовательности. Сначала слева направо выполняются входящие в его состав выражения, затем создается объект и выдается соответствующее значение. Однако для того чтобы реально не создавать объект, к которому не будет обращений в данном вызове программы, применяется следующая оптимизация. Если ⟨генератор⟩ является правой частью описания переменной или константы, то (генератор «запроцедуривается» и) его выполнение задерживается до момента первого считывания значения данной переменной или константы. *

*) В этом случае ⟨формирование⟩ не должно приводить к созданию адресного объекта (указатель, метка).

8. Присваивание

⟨Присваивание⟩ осуществляет замену текущего значения переменной на новое значение, вычисляемое в правой части.

⟨присваивание⟩ ::=

⟨переменная⟩ := ⟨выражение⟩

| ⟨переменная⟩ :=: ⟨выражение⟩

Если формат переменной и формат нового значения совпадают, то присваивание происходит без преобразования формата. При несоответствии форматов возможно преобразование формата значения:

Формат переменной (ФП)	Формат значения (ФЗ)	Преобразование
простой	простой	ФЗ>ФП — формат числового значения сокращается ФЗ<ФП — присваивание без преобразования
строчный или нестандартный	простой	ВРП значения обрезается слева до формата переменной

* Если требуется сокращение формата, а новое значение — объект нечислового типа, возникает ситуация *неверныйоперанд*. Сокращение формата целого от ф64 к ф32 может привести к преобразованию его в вещественное. Это происходит тогда, когда абсолютная величина исходного целого превышает величину максимально допустимого целого формата ф32. Сокращение формата вещественного до формата ф32 или ф64 может привести к возникновению ситуации *переполнвещ*.

В случае, если переменной формата ф32 присваивается набор, последний сначала преобразуется в целое ф64, а затем уже выполняется присваивание с сокращением формата целого.

Символом :=: обозначается целочисленное присваивание. В отличие от обычного присваивания здесь контролируется тип значения правой части. Тип должен быть целым, иначе возникает ситуация *неверныйоперанд*. Кроме того, в случае присваивания переменной формата ф32 контролируется переход целого формата ф64 в вещественное формата ф32. Если сокращение формата целого приводит к такому преобразованию, возникает ситуация *переполнцел32*.

Особые случаи. С целью оптимизации ⟨присваивание⟩ вида: $x.p := e$, где x — какая-либо ⟨переменная⟩, а e — ⟨выражение⟩, выполняется по аналогии с присваиванием $x := x.(p : e)$. В связи с этим возникают две особенности:

1. e не может производить побочный эффект, влияющий на значение x.

2. Если значение x имеет формат ф32, то нужно учитывать, что ВРП такого объекта имеет формат ф64, и при определении местоположения поля надо руководствоваться ВРП. Например,

пусть дана переменная y формата ф32. Установка знакового разряда в 1 программируется так: $y.[63] := 1$.

⟨Присваивание⟩, имеющее вид:

⟨указуемая переменная⟩ := e
⟨идентификатор⟩ формального := e
⟨закрытая переменная⟩ := e

где левая часть после выполнения определяет поле переменной, позволяет упаковать в это поле целое со знаком. Упаковка знака происходит в процессе присваивания: знаковый разряд исходного целого накладывается на старший разряд нового содержимого поля. Последующую распаковку целого, представленного таким образом, можно осуществить операцией целзн. *

⟨Присваивание⟩ может быть не только ⟨оператором⟩, но и ⟨выражением⟩. В последнем случае его значением является значение правой части в исходном (т. е. непреобразованном) виде.

9. Закрытые предложения

⟨Закрытые предложения⟩ служат для структурирования программы. Для этой цели введены четыре разновидности закрытых предложений:

1. Замкнутое предложение, играющее роль структурных скобок для последовательности описаний и предложений.

2. Условное и выбирающее предложения, которые дают возможность выбора альтернатив для выполнения.

3. Цикл, служащий для организации повторяющегося процесса вычислений.

4. Структурное предложение, являющееся средством структурированной передачи управления при обработке ошибок и других особых ситуаций.

Особенность синтаксического описания. ⟨Закрытое предложение⟩ может играть роль ⟨оператора⟩, может использоваться в ⟨выражении⟩ и в конструкциях, где требуется ⟨переменная⟩ (например — в левой части ⟨присваивания⟩). Структура ограничителей некоторого конкретного ⟨закрытого предложения⟩ не зависит от синтаксической позиции, в которой оно находится, но есть различие в том, какими предложениями может завершаться его выполнение. Чтобы не размножать число синтаксических правил по числу возможных позиций, используется параметризация некоторых правил, связанных с ⟨закрытыми предложениями⟩. Признаком параметризации является слово «предложение», встречающееся в левой и правой частях правила. Это слово можно рассматривать, как формальный параметр правила. Для того чтобы по-

лучить действующее правило, надо в обеих частях правила заменить слово «предложение» на одно из трех слов: «оператор», «выражение», «переменная». Например, из приведенного ниже правила для ⟨закрытого предложения⟩ образуются три правила: для ⟨закрытого оператора⟩, для ⟨закрытого выражения⟩ и для ⟨закрытой переменной⟩.

⟨закрытое предложение⟩ ::=
{блок} ⟨небазированное закрытое предложение⟩
⟨небазированный закрытый оператор⟩ ::=
⟨замкнутый оператор⟩ | ⟨условный оператор⟩
| ⟨выбирающий оператор⟩ | ⟨цикл⟩
| ⟨структурный оператор⟩
⟨небазированное закрытое выражение⟩ ::=
⟨замкнутое выражение⟩ | ⟨условное выражение⟩
| ⟨выбирающее выражение⟩ | ⟨структурное выражение⟩
⟨небазированная закрытая переменная⟩ ::=
⟨замкнутая переменная⟩ | ⟨условная переменная⟩
| ⟨выбирающее переменную⟩

* *Базированный блок.* Процедура и ⟨закрытое предложение⟩, начинающееся словом **блок**, называются базированными блоками. В начале выполнения базированного блока создается связанная с ним базированная область памяти (БОП). В БОП отводится память для переменных и констант, создаваемых с помощью ⟨описаний⟩, и в ряде случаев — память для составных объектов. Если слово **блок** отсутствует, то отводимая память располагается в текущей БОП. При этом распределение ячеек ведется так, чтобы обеспечить максимальное наложение памяти, используемой непересекающимися небазированными блоками. *

Последовательное предложение. Эта конструкция является компонентой ⟨замкнутого предложения⟩ и альтернативных предложений (условного, выбирающего, структурного). Оно используется тогда, когда ограничители этих предложений могут играть роль структурных скобок для заключенной между ними последовательности действий.

⟨последовательное предложение⟩ ::=
{⟨описание⟩;}...{⟨помеченный оператор⟩;}...
⟨помеченное предложение⟩
⟨помеченное предложение⟩ ::= {⟨метка⟩}...⟨предложение⟩
⟨оператор⟩ ::=
⟨присваивание⟩ | ⟨модификация атрибутов⟩
| ⟨вызов⟩ | ⟨генератор внешнего объекта⟩
| ⟨структурный переход⟩ | ⟨запуск задачи⟩
| ⟨операции над массивами⟩ | ⟨подстановка текста⟩
| ⟨закрытый оператор⟩ | ⟨двоичный обмен⟩

| ⟨форматный обмен⟩ | ⟨формирование паспорта⟩
| ⟨переход⟩

⟨Последовательное предложение⟩ задает прямую последовательность действий. Сначала выполняются ⟨описания⟩, затем ⟨операторы⟩ и в заключение — ⟨предложение⟩. Областью действия ⟨описаний⟩ является данное ⟨последовательное предложение⟩.

Результат выполнения. Значением ⟨последовательного выражения⟩ является значение завершающего ⟨выражения⟩. ⟨Последовательная переменная⟩ обозначает переменную, определяемую в результате выполнения завершающей ⟨переменной⟩. Этот результат, в свою очередь, является результатом объемлющей конструкции (⟨закрытого выражения⟩ или ⟨закрытой переменной⟩).

9.1. Замкнутое предложение. ⟨Замкнутое предложение⟩ представляет собой простейший способ структурирования описаний и предложений. Оно используется в основном для организации блоков с локальными описаниями (в частности для оформления тела процедуры), а также для изменения порядка вычислений в ⟨формулах⟩. ⟨Замкнутое предложение⟩ представляет собой ⟨последовательное предложение⟩, заключенное в скобках: (и), или **начало** и **конец**.

⟨замкнутое предложение⟩ ::=
(⟨последовательное предложение⟩)
| **начало** ⟨последовательное предложение⟩ **конец**

Пример. Блок с описанием.

```
начало ф128 c := вещ128 0 ;
    для i до вв цикл
        c := c + в128 f(i) ** 2
    повторить
конец
```

Пример. ⟨Замкнутое предложение⟩ в роли ⟨выражения⟩.

$a := (b + c) * d$

9.2. Условное предложение. Условное предложение позволяет из нескольких альтернатив выбрать и выполнить ту, для которой ⟨условие⟩ принимает значение **истина**.

⟨условное предложение⟩ ::=
если ⟨условие⟩ **то** ⟨последовательное предложение⟩
{**иначе** ⟨условие⟩ **то** ⟨последовательное предложение⟩}...

{**иначе** ⟨последовательное предложение⟩}
все

⟨условие⟩ ::=
{⟨описание⟩;} ... {⟨помеченный оператор⟩;} ...
⟨помеченное выражение⟩

При выполнении ⟨условного предложения⟩ последовательно выполняются ⟨условия⟩ до первого, ⟨выражение⟩ которого принимает значение **истина**. Затем выполняется соответствующее ⟨последовательное предложение⟩. Если такого ⟨условия⟩ не нашлось, но ⟨условное предложение⟩ содержит альтернативу **иначе**, то выполняется ⟨последовательное предложение⟩ из этой альтернативы. Если же ни одно ⟨условие⟩ не выполнилось и нет альтернативы **иначе**, то ни одно ⟨последовательное предложение⟩ не выполняется. ⟨Условное выражение⟩ и ⟨условная переменная⟩ с необходимостью должны содержать альтернативу **иначе**.

* *Уточнение относительно значения ⟨условия⟩.* Значение ⟨условия⟩ может иметь, вообще говоря, любой тип. Формат не должен превышать формата ф64. При проверке условия учитывается только содержимое правого крайнего разряда в ВРП значения. Если оно равно 1(0), то считается, что условие принимает значение **истина** (**ложь**). *

⟨Условие⟩ может содержать ⟨описания⟩. Областью действия каждого такого описания является вся следующая за ним часть ⟨условного предложения⟩, вплоть до закрывающего ограничителя **все**.

Пример.

```
если месяц = 12 и послдень (год, месяц, день) то
    месяц := 1;
    день := 1;
    год := год + 1
иначе послдень (год, месяц, день) то
    месяц := месяц + 1;
    день := 1
иначе день := день + 1
все
```

Пример. Сокращенная запись предыдущего примера.

```
день := если послдень (год, месяц, день) то
месяц := если месяц = 12 то
            год := год + 1;
            1                  % первый месяц след. года
         иначе месяц + 1 % следующий месяц
         все;
```

```
    1                          % первый день след. месяца
  иначе день + 1               % следующий день
  все
```

Процедура *последень* описана в примере п. 9.3.

9.3. Выбирающее предложение. ⟨Выбирающее предложение⟩ позволяет выбрать одну из пронумерованных альтернатив и выполнить ее. Номер выполняемой альтернативы вычисляется в заголовке ⟨выбирающего предложения⟩.

⟨выбирающее предложение⟩ ::=
выбор ⟨вычисление номера **из**
⟨список размеченных альтернативных предложений⟩
{**иначе** ⟨последовательное предложение⟩}
всевыб
| **выбор** ⟨вычисление номера⟩ **из**
⟨список последовательных предложений⟩
{**иначе** ⟨последовательное предложение⟩}
всевыб

⟨вычисление номера⟩ ::=
{⟨описание⟩;} ... {⟨помеченный оператор⟩;} ...
⟨помеченное выражение⟩

⟨размеченное альтернативное предложение⟩ ::=
{⟨номер⟩:,} ... ⟨номер⟩:⟨последовательное предложение⟩

⟨номер⟩ ::= ⟨выражение⟩

Альтернативы ⟨выбирающего предложения⟩ нумеруются либо в явной форме, либо неявно. В первом случае ⟨выбирающее предложение⟩ строится из ⟨размеченных альтернативных предложений⟩. Каждой альтернативе можно приписать один или несколько ⟨номеров⟩. ⟨Номер⟩ — это выражение статического класса, значением которого может быть целое или набор.

Во втором случае номер в явном виде отсутствует, и считается, что альтернативы пронумерованы от 0 до $n-1$, где n — число альтернатив.

Выполнение ⟨выбирающего предложения⟩ происходит следующим образом. Выполняется ⟨вычисление номера⟩. Значение ⟨выражения⟩ определяет номер альтернативы, которую следует выполнить. Если такая альтернатива существует, то выполняется ее ⟨последовательное предложение⟩. Если же такой альтернативы нет, но есть альтернатива **иначе**, то выполняется последовательное предложение из альтернативы **иначе**. В противном случае ни одно ⟨последовательное предложение⟩ не выполняется. ⟨Выбирающее выражение⟩ и ⟨выбирающее переменную⟩ обязательно должны содержать альтернативу **иначе**.

Примеры.

```
выбор день из
1 : , 3 : , 7 :   печать (стр8''дневная смена'') ,
2 : , 4 : печать (стр8''вечерняя смена'') ,
5 : , 6 : печать (стр8''выходной день'')
иначе печать (стр8''неверно задан день'')
всевыб

процедура последень = функция (год , месяц , день)
{день = выбор месяц из
        1 : , 3 : , 5 : , 7 : , 8 : , 10 : , 12 : 31 ,
        4 : , 6 : , 9 : , 11 : 30 ,
        2 :    если год остат 4 = 0 и
                  год остат 100 <> 0 или
                  год остат 400 = 0 то
                29 %     високосный год
                иначе    28 все
      иначе 0 всевыб)

приоритет : =
   выбор лексема из
   "или" : 2 , "и" : 3,          % логические операции
   "<" : , ">" : , "=" : , "<=" : , ">=" : , "<>" : 4 ,
                                 % операции отношения
   "+" : , "—" : 5 ,             % операции типа сложения
   "*" : , "/" : 6 ,             % операции типа умножения
   "**" : 7                      % возведение в степень
   иначе
      ошибка (стр8''нет такой операции'')
   всевыб
```

9.4. Цикл. Конструкция ⟨цикл⟩ предназначена для организации повторяющегося процесса выполнения ее ⟨операторов⟩.

```
⟨цикл⟩ ::=
    {⟨заголовок цикла⟩}
     цикл
    {⟨описание⟩ ;}...
    {⟨помеченный оператор⟩ ; }...
     ⟨помеченный оператор⟩
     повторить
```

Повторяющееся выполнение ⟨операторов⟩ цикла без заголовка может быть прекращено с помощью ⟨структурного пе-

рехода⟩, пересекающего границы цикла (и, кроме того, с помощью обычного перехода на метку).

Цикл с заголовком определяет, что ⟨операторы⟩ выполняются столько раз, сколько задано в ⟨заголовке цикла⟩.

⟨заголовок цикла⟩ ::=
для ⟨идентификатор⟩ [**от** ⟨выражение⟩] {⟨компонента-до⟩}
| **от** ⟨выражение⟩ {⟨компонента-до⟩}
⟨компонента-до⟩ ::= **до**⟨выражение⟩ | **вниздо** ⟨выражение⟩

В ⟨заголовке цикла⟩ задается ⟨идентификатор⟩ параметра цикла. Появление этого идентификатора в заголовке одновременно является и его описанием в пределах данного ⟨цикла⟩. Начальное значение параметра цикла задается ⟨выражением⟩ в компоненте **от**, а конечное значение — выражением в ⟨компоненте-до⟩. Ограничитель ⟨компоненты-до⟩ задает направление изменения параметра цикла: на каждой итерации его значение увеличивается на 1 в случае ограничителя **до** и уменьшается на 1 в случае ограничителя **вниздо**.

Семантика выполнения цикла с заголовком отображается следующей программой:

```
... установить начальное значение i ... ;
до кц
цикл
    если (i — u) * s > 0 то кц! все ;
    ... выполнить операторы цикла ...;
    ... увеличить (уменьшить) i на 1 ...
повторить
```

где i — параметр цикла (начальное значение задается в заголовке), u — конечное значение параметра цикла, $s = 1$ для цикла по возрастанию значения параметра и $s = -1$ для цикла по убыванию значения параметра.

* Начальное и конечное значение параметра цикла вычисляется один раз при входе в цикл. Значения ⟨выражений⟩ должны быть целыми числами. В ⟨выражениях⟩ ⟨заголовка цикла⟩ нельзя использовать ⟨идентификатор⟩ параметра того же цикла. Если компонента **от** и/или ⟨компонента-до⟩ опущены, то начальное и/или конечное значение выбираются по умолчанию. Умолчание для начального значения —0, а для конечного значения $-2^{20} - 1$.

Параметр цикла на каждой итерации рассматривается как константа. Его значение нельзя изменять с помощью присваивания, его нельзя использовать в качестве параметра, передаваемого именем, и в качестве объекта при формировании указателя.

Идентификатор параметра цикла действует только внутри цикла. Однако, если выполнение цикла прекращается ⟨структурным переходом⟩, то текущее значение параметра можно передать в качестве фактического параметра реакции.

Пример. Найти значение x в векторе $a[0; n+1]$

```
начало
    a[n] := x ;
    ix := до найден
        для i до n
        цикл
            если a [i] = x то найден ! (i) все
        повторить
        при найден (k) : если k = n то —1 иначе k все
        всесит
конец
```

Компоненту для можно опускать в том случае, если в ⟨операторах⟩ цикла параметр цикла не используется. *

⟨Описания⟩ из тела ⟨цикла⟩ выполняются один раз при входе в цикл. Областью действия каждого такого описания является вся последующая часть ⟨цикла⟩, вплоть до закрывающего ограничителя повторить.

Пример. Умножение матриц a и b размера $n \times n$.

```
для i до n — 1
цикл ф128 cij ; % промежуточные вычисления проводятся
                % с двойной точностью
    для j до n — 1
    цикл
        cij := 0 ;
        для k до n — 1
        цикл
            cij := cij + a [i ,k]умнд в [k , j]
        повторить
        c [i , j] := вещ64окр cij % округление результата
    повторить
повторить
```

10. Обработка ситуаций

Семантика конструкций описывает нормальный процесс их выполнения. В частности, под термином «завершение» понимается нормальное окончание выполнения конструкции. В отличие от этого в данном разделе описываются средства языка, предназначенные для программирования алгоритмов обработ-

ки ошибок и других особых обстоятельств, при которых прекращается процесс нормального выполнения предложений программы. Эти обстоятельства объединяются под общим названием «ситуации». О возникновении ситуации сигнализирует конструкция ⟨структурный переход⟩, которая (без учета возможного списка параметров) имеет вид:

⟨первичное⟩!

⟨Первичное⟩ задает возникшую ситуацию. ⟨Структурный переход⟩ прекращает выполнение ⟨закрытого предложения⟩, управляющего данной ситуацией. Говорят, что ⟨закрытое предложение⟩ управляет ситуацией s в том случае, если оно входит в состав ⟨структурного предложения⟩, в заголовке которого указана данная ситуация s. ⟨Структурное предложение⟩ имеет вид:

до ⟨список определения ситуаций⟩

⟨закрытое предложение⟩

{приставка альтернативных предложений}

Например, цикл, входящий в состав ⟨структурного предложения⟩,

```
до найденнуль
цикл
    читф(вхфайл, имя x); % ввести следующее число
    если x = 0 то найденнуль! все ;
    сумма := сумма + x
повторить
```

управляет ситуацией *найденнуль*. При возникновении этой ситуации выполнение цикла прекращается.

Прекращение выполнения, в отличие от завершения, означает, что пропускаются все действия от точки возникновения ситуации до конца соответствующего ⟨закрытого предложения⟩ (в приведенном примере оператор присваивания на последней итерации не выполняется). В заключение выполняется реакция на ситуацию, если она была предусмотрена в «приставке альтернативных предложений».

10.1. Структурное предложение. ⟨Структурное предложение⟩ служит для того, чтобы выделить ⟨закрытое предложение⟩, управляющее определенными ситуациями. Таких ситуаций может быть несколько. Все они определяются в заголовке ⟨структурного предложения⟩. После заголовка следует выделяемое ⟨закрытое предложение⟩

⟨структурное предложение⟩ ::=
до ⟨список определений ситуаций⟩
⟨закрытое предложение⟩
{**при** ⟨список альтернативных предложений⟩ **всесит**}

В конструкции может быть задан ⟨список альтернативных предложений⟩, в котором указываются реакции на ситуации, определенные в заголовке. Если список полностью опущен или же в нем указаны реакции не для всех ситуаций, то это означает, что соответствующие реакции выбираются по умолчанию.

* Если ⟨закрытое предложение⟩, входящее в состав ⟨структурного предложения⟩ с приставкой, в свою очередь, является ⟨структурным предложением⟩, то оно также должно содержать приставку. *

Выполнение ⟨структурного предложения⟩ завершается либо после нормального завершения его ⟨закрытого предложения⟩, либо после нормального завершения реакции на ситуацию.

Закрытое предложение может управлять как статической, так и динамической ситуацией.

⟨определение ситуации⟩ ::= ⟨идентификатор⟩
| *⟨идентификатор⟩

Статической ситуацией управляет только одно ⟨закрытое предложение⟩. Связь предложения и ситуации задается одним из двух способов:

— с помощью ⟨определения ситуации⟩ в форме: ⟨идентификатор⟩. Такая форма определения является описанием, которое вводит статическую ситуацию и связывает с ней ⟨идентификатор⟩. Это описание действует в пределах данного ⟨структурного предложения⟩;

— с помощью ⟨определения ситуации⟩ в форме: * ⟨идентификатор⟩, где ⟨идентификатор⟩ обозначает ситуацию, описанную ранее с помощью предварительного ⟨описания статической ситуации⟩.

⟨описание статических ситуаций⟩ ::=
статсит ⟨список идентификаторов⟩

* Предварительное описание предназначено для того, чтобы описание процедур, в которых возникает эта ситуация, могло текстуально предшествовать ее определению в ⟨структурном предложении⟩.

Пример.

```
начало
статсит s ;
процедура p = проц (...) начало ...s ! ... конец ;
  ...
```

```
до * s
начало
... p (...) ...
конец
...
конец
```

Предварительное описание вводит статическую ситуацию и связывает с ней ⟨идентификатор⟩. Одно описание может вводить несколько ситуаций. Такое описание имеет обычную область действия. В этой области действия ситуацией может управлять лишь одно ⟨закрытое предложение⟩. *

Динамической ситуацией, в отличие от статической, могут одновременно управлять несколько ⟨закрытых предложений⟩. В частности, допускается случай, когда в процессе выполнения программы эти предложения оказываются динамически вложенными одно в другое.

Динамическая ситуация вводится не с помощью статического описания, а создается динамически в результате выполнения ⟨генератора ситуации⟩.

⟨генератор ситуации⟩ ::=

ситуация({⟨список установки атрибутов⟩})

Чтобы в ⟨списке определения ситуаций⟩ указать динамическую ситуацию, надо использовать форму: *⟨идентификатор⟩, где ⟨идентификатор⟩ обозначает константу или переменную, текущим значением которой является динамическая ситуация.

* Ситуация имеет два атрибута: *процкр* и *локал*. Значением атрибута *процкр* является процедура конечной реакции на ситуацию. Если этот атрибут опущен, то считается, что конечная реакция не предусмотрена.

Значение атрибута *локал* определяет локализацию ситуации: **истина** означает, что этот объект является локальным, а **ложь** — глобальным. По умолчанию ситуация считается локальной.

В стандартном контексте (см. § 12) описаны константы, значениями которых являются стандартные динамические ситуации, связанные с ошибками вычислений, с ошибками, которые могут возникнуть в процессе взаимодействия с внешними объектами и др. Любое предложение программы может взять на себя управление такой ситуацией. Например:

```
до * ситлкф
цикл
    ... чит (вхтекст , ...)
повторить
при
    ситлкф : ... R ...
всесит
```

Здесь чтение файла *вхтекст* продолжается до тех пор, пока не встречен логический конец файла (соответствующая ситуация является значением глобальной константы *ситлкф*), после чего выполнение цикла прекращается и предпринимается реакция R. Обстоятельства, при которых возникают стандартные ситуации, описаны в соответствующих разделах семантики. Идентификаторы стандартных ситуаций перечислены в Приложении 6. *

Реакция на ситуацию. Приставка ⟨структурного предложения⟩ состоит из ⟨альтернативных предложений⟩, описывающих реакции на ситуации, которыми управляет ⟨закрытое предложение⟩, входящее в состав данного ⟨структурного предложения⟩.

⟨альтернативное предложение⟩ ::=
{⟨идентификатор⟩:,} ... ⟨идентификатор⟩
{({⟨описание формальных⟩;} ... ⟨описание формальных⟩)}
: ⟨последовательное предложение⟩
⟨описание формальных⟩ ::= {⟨простой формат⟩}
⟨список идентификаторов⟩

⟨Идентификаторы⟩, находящиеся в начале альтернативы, перечисляют ситуации, для которых предусмотрена данная реакция. Для обозначения статической или динамической ситуации используются те же идентификаторы, что и в заголовке данного ⟨структурного предложения⟩. ⟨Последовательное предложение⟩ задает действия, которые надо выполнить в качестве реакции.

В ⟨альтернативное предложение⟩ могут входить ⟨описания формальных⟩, предназначенные для параметризации реакции. Каждый формальный параметр эквивалентен простой переменной, которой в момент инициирования реакции присваивается начальное значение, равное значению соответствующего фактического параметра структурного перехода. Если ⟨простой формат⟩ опущен, то подразумевается, что формат параметра — ф64.

В случае, если для какой-либо ситуации, определенной в заголовке, реакция явно не задана (в частности, если приставка полностью опущена), она выбирается по умолчанию. Умолчание определяется следующим образом:

— в случае ⟨структурного оператора⟩ реакция представляет собой пустое действие;

— в случае ⟨структурного выражения⟩ выдается значение фактического параметра ⟨структурного перехода⟩. Это свойство можно, в частности, использовать при программировании процедур-функций. Например, выполнение функции

```
функция (...)
    до резвычислен
    начало
        ...
        резвычислен ! (е) ;
        ...
    конец
```

прекращается при возникновении ситуации *резвычислен*. Результирующим значением является значение выражения *е*.

* В реакции ⟨структурного предложения⟩ нельзя использовать статическую ситуацию, определенную в его заголовке (так как это приведет к зацикливанию). В отличие от этого динамическую ситуацию можно использовать. В частности, ее можно использовать в ⟨структурном переходе⟩ из реакции. При этом необходимо учитывать, что в момент инициирования реакции восстанавливается смысл, приданный динамическим ситуациям в ⟨структурных предложениях⟩, охватывающих данное (см. п. 10.2). *

10.2. Структурный переход. ⟨Структурный переход⟩ сигнализирует о возникновении ситуации.

⟨структурный переход⟩ ::= ⟨первичное⟩!
{(⟨список выражений⟩)}

⟨Структурный переход⟩ по ситуации *s*, являющийся значением ⟨первичного⟩, прекращает выполнение группы предложений и процедур, динамически охватывающих ⟨структурный переход⟩. В эту группу входят все предложения и процедуры, начиная с ближайшего, охватывающего ⟨структурный переход⟩, и кончая тем ⟨закрытым предложением⟩, которое управляет ситуацией *s*.

* Прекращение выполнения блока (в том числе процедуры), как и его завершение, приводит к тому, что локализованные в нем объекты ликвидируются. *

В заключение выполняется реакция на ситуацию *s*, указанная в том ⟨структурном предложении⟩, куда произошел возврат.

В случае динамической ситуации имеет место следующая особенность. Если ⟨структурный переход⟩ охвачен одновременно несколькими ⟨закрытыми предложениями⟩, управляющими одной и той же ситуацией *s*, то прекращается выполнение только ближайшего из них.

Пример.

```
начало
    конст s = ситуация (    ,
          p = проц (x) начало ... s ! ...конец ;
    ...
    до * s
    начало % блок A
        ..
        до * s
        начало % блок B
            . .
            p (z) ;
            ...
        конец
        при
        s : ... RB ...
        всесит ;
        ...
        p (y) ;
        ...
    конец
    при
    s : .. RA ...
    всесит ;
    ...
конец
```

Здесь может произойти следующее.

1. Если ситуация s возникает в процессе выполнения вызова $p(z)$, то прекращается выполнение процедуры p и блока B, затем выполняется реакция RB; после этого выполняются операторы, следующие за ⟨структурным предложением⟩ B.

2. Если ситуация s возникает при вызове $p(y)$, то прекращается выполнение процедуры p и блока A, затем выполняется реакция RA; после этого выполняются операторы, следующие за ⟨структурным предложением⟩ A.

* *Конечная реакция.* В процессе выполнения ⟨структурного перехода⟩ по динамической ситуации может оказаться, что в данном независимом процессе (см. п. 11.2) не существует охватывающего ⟨закрытого предложения⟩, управляющего этой ситуацией. В этом случае прекращается выполнение независимого процесса, в котором возникла ситуация. Затем выполняется конечная реакция, заданная при генерации ситуации. Если конечная реакция не предусмотрена, выдается сообщение «не переопределена ситуация». В заключение независимый процесс ликвидируется.

Необходимо учитывать, что к началу выполнения конечной реакции оперативные объекты (массивы, оперативные объекты связи с внешними объектами) и внешние объекты, локализованные в прекращенном процессе, уже ликвидированы. По этой причине подобные объекты нельзя использовать в конечной реакции. *

Параметры. Конструкция может содержать ⟨список выражений⟩, значения которых передаются в реакцию (в частности,— в конечную реакцию) в качестве фактических параметров.

Пример. Передача параметров в реакцию.

```
для i от 2 до n
цикл % сортировка массива a методом простой
     % вставки; значение элемента a [0] является
     % ограничителем при поиске
   ф64 x;
   x := a [0] : =a [i];
   до найденеменьшех
   для j от i — 1 вниздо 0
   цикл
      если a [j] <= x то найденеменьшех ! (j)
      иначе a [j + 1] := a [j] % сдвиг
      все
   повторить
   при найденеменьшех (к) : a [к + 1] := x
   всесит
повторить
```

Пример. Использование стандартных ситуаций.

```
процедура факториал = функция (n)
   до * неверныйоперанд ,* переполнцел64
   если n <= 1 то 1
   иначе n *: факториал (n — 1
   все
   при неверныйоперанд : , переполнцел64: максцел
   всесит
```

В этом примере при выполнении целочисленного умножения (см. п. 7.5) могут возникнуть стандартные ситуации *неверныйоперанд* (если тип операнда отличен от целого) и *переполнцел64* (при переполнении целочисленного результата). В обоих случаях функция выдает максимальное целое, что приведет к возникновению ситуации *переполнцел64* в предыдущем поколении активации функции и т. д. В заключение первое поколение активации выдаст максимальное целое.

11. Единицы вычислений

В этом параграфе описываются три единицы вычислений: выполнение процедуры, процесс и задача.

11.1. Процедура. Базовой конструкцией языка, предназначенной для использования процедурного механизма, является ⟨текст процедуры⟩, имеющий вид:

⟨спецификация результата⟩(⟨параметры⟩)
⟨закрытое предложение⟩

Эта конструкция задает собственно содержание процедуры (спецификацию результата, параметры и описание алгоритма в виде ⟨закрытого предложения⟩), но не связывает процедуру с идентификатором. Значением ⟨текста процедуры⟩ является метка процедуры (далее будем говорить не «метка процедуры», а просто «процедура»).

Средства работы с таким значением обычные: оно может быть присвоено переменной, передано в качестве фактического параметра, сделано значением константы. Например, если дана переменная *p* формата ф64, то в результате ⟨присваивания⟩

p := **проц** (...) **начало** ... **конец**

ее значением становится процедура, изображенная в правой части. Пока процедура является значением *p*, можно считать, что *p* — это идентификатор данной процедуры.

При необходимости можно создать неразрывную связь между процедурой и идентификатором. Это делается путем задания ⟨текста процедуры⟩ в правой части ⟨описания простой константы⟩.

Пример.

```
конст скапр = функция (a , b)
    начало ф128 сум := вещ128 0 ;
        для i до (длина a — 1)
        цикл
            сум := сум + a [i] умнд b [i]
        повторить ;
        вещ64окр сум
    конец
```

⟨Спецификация результата⟩ может иметь вид **проц** или **функция**. Первая задает процедуру, в результате выполнения которой значение не выдается, а вторая — процедуру-функцию.

⟨Вызов⟩ процедуры имеет вид:

⟨первичное⟩ (... фактические параметры ...)

где значение ⟨первичного⟩ — это вызываемая процедура. Например, если значением переменной *p* и константы *скапр* являются процедуры, то их вызов обозначается следующим образом:

p (.. фактические параметры ...)
скапр (.. фактические параметры ...)

Существует четыре способа передачи параметров: «значением», «именем», «значение подпрограммой», «имя подпрограммой» (см. ниже).

Текст процедуры. Правило для ⟨текста процедуры⟩ имеет следующий вид:

⟨текст процедуры⟩ ::=
проц ({⟨параметры⟩})⟨закрытый оператор⟩
| **функция** {⟨параметры⟩})⟨закрытое выражение⟩
⟨параметры⟩ ::=
{⟨описание формальных процедуры⟩ ; } ...
⟨описание формальных процедуры⟩
⟨описание формальных процедуры⟩ ::=
⟨описание формальных⟩ | ⟨описание базы⟩

Спецификация результата. Спецификация **проц** задает процедуру, в результате выполнения которой значение не выдается. Такую процедуру нельзя вызвать в ⟨выражении⟩ (точнее, вызов этой процедуры не может являться ⟨первичным⟩).

Спецификация **функция** задает процедуру-функцию. В результате выполнения функции выдается значение ее ⟨закрытого выражения⟩. Например, выполнение приведенной в предыдущем разделе процедуры *скапр* завершается вычислением формулы **вещ64окр** *сум*, результат которой является значением этой функции. Вызов процедуры-функции может выполнять роль ⟨первичного⟩ и роль ⟨оператора⟩.

Формальные параметры. В ⟨тексте процедуры⟩ могут быть заданы описания формальных параметров. Эти описания аналогичны по форме описаниям простых переменных. Как и в случае переменных, задается только формат параметра, но не тип значения. По умолчанию выбирается формат ф64. Областью действия формальных параметров является ⟨закрытое предложение⟩.

Для процедуры без параметров за спецификацией результата следует пара скобок: ( ).

* При вызове процедуры образуется базированный блок, у которого в начале БОП располагаются фактические параметры. Если ⟨закрытое предложение⟩ является небазированным бло-

ком, то память для локальных описаний отводится следом в той же самой БОП. В противном случае для ⟨закрытого предложения⟩ создается новая БОП.

⟨Текст процедуры⟩ с переменным числом параметров может быть организован следующим образом:

⟨спецификация результата⟩ (... ; **база** *перемпарам*)
блок ... *перемпарам* [*к*] ...

Здесь тело процедуры является базированным блоком; ⟨описанию базы⟩ предшествует описание фиксированной части параметров (если такая есть). При вызове такой процедуры нужно сначала указывать фиксированную часть фактических параметров, а затем переменную часть. Значением *перемпарам* является указатель, описывающий переменную часть параметров как с-вектор элементов формата ф32 (см. § 16); *перемпарам* [*к*] обозначает *к*-й элемент. Например, ⟨текст процедуры⟩ вывода чисел может иметь такой вид:

```
конст печатьчисел = проц (файл ; база числа)
    блок
    начало
      конст числаф64 = числа [ф64 :] ; % преобразование
                                        % формата
      ...
      для к до (длина числаф64 — 1)
      цикл
      ... числаф64 [к] ... % обращение к к-му параметру
                           % переменной части
      повторить ;
      ...
    конец
```

В списке фактических параметров этой процедуры на первом месте надо указывать файл; за ним следует список любой длины, задающий выводимые числа:

печатьчисел(*файлацпу*, *е1*, *е2*, ..., *еп*) *

При вызове процедуры фактические параметры ставятся в соответствие формальным. Способ передачи фактического параметра задается в месте вызова. Он определяет:

1. Вычисляется ли фактический параметр один раз при входе в процедуру, или же каждый раз при обращении к соответствующему формальному параметру.

2. Что ставится в соответствие формальному параметру — переменная, являющаяся фактическим параметром, или ее значение.

Способ передачи параметров может быть различным в различных вызовах одной и той же процедуры (см. ниже).

* Локализация процедуры. Процедура локализована в минимальном из двух ⟨закрытых предложений⟩, охватывающих ⟨текст процедуры⟩:

— ближайшем, содержащем описания объектов, использованных внутри процедуры,

— ближайшем базированном блоке.

Процедуру нельзя вызывать после окончания выполнения ⟨закрытого предложения⟩, в котором она была локализована. Это ограничение не распространяется на процедуры, реализующие конечные реакции на динамические ситуации. *

Описание процедур-констант. ⟨Описание процедур-констант⟩ предназначено для рекурсивного и взаимно рекурсивного определения процедур.

⟨описание процедур-констант⟩ ::=
процедура ⟨список идентификации процедур⟩
⟨идентификация процедуры⟩ ::=
⟨идентификатор⟩{=⟨текст процедуры⟩}

⟨Описание процедур-констант⟩ вводит один или несколько ⟨идентификаторов⟩ констант, обозначающих процедуры. В правой части ⟨идентификации процедуры⟩ разрешается рекурсивное использование ее ⟨идентификатора⟩, например:

процедура *нод* = **функция** (*a* , *b*)
если *b* = 0
то *abs* (*a*)
иначе *нод* (*b* , *a* остат *b*)
все

Если правая часть ⟨идентификации процедуры⟩ опущена, то такое описание является предварительным. Оно используется для взаимно рекурсивного определения процедур, например:

процедура *p*, *q*;
процедура
p = **проц** (...) **начало** ... *q* ... **конец**,
q = **проц** (...) **начало** ... *p* ... **конец**;

Предварительное описание процедуры (если такое есть) и ее непосредственное описание должны находиться в одной и той же последовательности ⟨описаний⟩, причем предварительное описание должно предшествовать непосредственному.

Вызов процедуры. Правило для ⟨вызова⟩ процедуры имеет следующий вид:

⟨вызов⟩ ::= ⟨первичное⟩ ({⟨список фактических⟩})
⟨фактический⟩ ::=
{**пак32**} ⟨выражение⟩ | **прог** ⟨выражение⟩
| **имя** ⟨переменная⟩ | **имя прог** ⟨переменная⟩

Эта конструкция выполняет вызов процедуры, являющейся значением ⟨первичного⟩. При каждом вызове устанавливается соответствие между фактическими и формальными параметрами, и затем в контексте ⟨текста процедуры⟩ выполняется ее ⟨закрытое предложение⟩. При необходимости процедура-функция в заключение выдает значение.

Передача параметров. Первая альтернатива правила для ⟨фактического параметра⟩ соответствует случаю, когда параметр передается «значением». Значение фактического параметра вычисляется при вызове процедуры. Это значение рассматривается как начальное значение соответствующего формального параметра.

* Число и формат значений фактических параметров должны совпадать с числом и форматом соответствующих формальных параметров (если не предпринято специальных мер по организации процедуры с переменным числом параметров, см. выше). В противном случае в процессе выполнения процедуры может возникнуть ситуация *ошадреса*. Для формального параметра формата ф64 допускается, чтобы значение фактического параметра имело формат ф32. В случае формального параметра формата ф32 необходимо указывать в фактическом параметре служебное слово пак32.

В случае процедуры с переменным числом параметров указанное ограничение распространяется на постоянную часть параметров. В связи с тем, что значения параметров передаются без какого-либо преобразования их формата, может оказаться, что область памяти, содержащая переменную часть, заполнена неоднородно. Это необходимо учитывать при выборе способа индексации вектора переменной части.

Существует три дополнительных способа передачи параметров:

1. Значение подпрограммой (**прог**). Значение фактического параметра вычисляется при каждом обращении к соответствующему формальному параметру в процессе выполнения процедуры. Вычисление происходит в контексте точки вызова. Формальный параметр может использоваться только в качестве ⟨первичного⟩ (но не ⟨переменной⟩).

2. Именем (**имя**). При входе в процедуру переменная, переданная в качестве фактического параметра, ставится в соответствие формальному параметру. Последний может использоваться внутри процедуры как ⟨переменная⟩ и как ⟨первичное⟩. Каждое обращение (для выборки или для присваивания) к формальному параметру в процессе выполнения процедурой означает обращение к соответствующей фактической переменной.

3. Имя подпрограммой (**имя прог**). Отличается от способа «именем» тем, что фактический параметр выполняется не один раз при входе в процедуру, а при каждом обращении к соответствующему формальному параметру в процессе работы процедуры. Выполнение происходит в контексте точки вызова.

В каждом из этих трех способов требуется, чтобы формат формального параметра был словным (ф64). При этом допускается любой формат фактического параметра.

11.2. Параллельные процессы и синхронизация. Выполнение процедуры может происходить в синхронном и параллельном режимах. В первом случае вызывающая процедура передает управление вызываемой и сама приостанавливается; после завершения выполнения вызываемой управление возвращается в точку вызова и выполнение вызывающей возобновляется. Такая схема управления реализуется конструкцией ⟨вызов⟩.

В случае параллельного режима вызывающая процедура лишь активизирует выполнение вызываемой, после чего продолжается выполнение предложений, следующих за тем, которое произвело активизацию. Параллельно с этим происходит выполнение вызванной процедуры.

Действие по активизации выполнения в параллельном режиме называется созданием параллельного процесса. Процедура, вызванная в таком режиме, называется головной процедурой процесса. Головная процедура может в свою очередь вызывать другие процедуры в синхронном или параллельном режиме.

Процесс завершается при завершении его головной процедуры. Кроме нормального завершения возможно прекращение процесса. Оно происходит тогда, когда выполнение головной процедуры прекращается из-за распространения ситуации.

Подчиненные и независимые процессы. Различаются два основных класса процессов: независимые и подчиненные.

В первом случае не существует какой-либо неявной синхронизации между независимым процессом и процессом, его породившим. В частности, если нет явно заданной синхронизации, то окончание (нормальное завершение или прекращение) независимого процесса не связано с какими-либо моментами выполнения создавшего процесса. Если прекращение произошло вследствие распространения ситуации, то перед ликвидацией процесса выполняется конечная реакция, связанная с данной ситуацией (см. п. 10.2). Если независимый процесс прекратился из-за перехода на глобальную метку, то выдается сообщение об ошибке.

В случае подчиненного процесса (*B*) существует критический блок в создавшем процессе (*A*), охватывающий место запуска *B*. Выполнение критического блока заканчивается не раньше, чем заканчивается выполнение *B*. Если выполнение всех предложений критического блока нормально завершилось до того, как закончился процесс *B*, то процесс *A* ожидает окончания *B*. Выполнение критического блока заканчивается лишь после того, как закончилось выполнение *B*. Поэтому подчиненный процесс может использовать переменные, константы и объекты, локализованные в критическом блоке (и блоках, охватывающих критический), не предпринимая при этом специальных мер по синхронизации.

При нормальном завершении процесса *B* критический блок продолжает выполняться в случае, если еще остались не выполненные предложения или неокончные подчиненные процессы.

Если выполнение критического блока прекращается из-за распространения ситуации, то одновременно прекращается вы-

полнение процесса *B* (и других подчиненных процессов этого блока); затем ситуация продолжает распространяться за пределами критического блока в процессе *A*.

Если выполнение процесса *B* прекращается из-за распространения ситуации, то дальнейшее распространение данной ситуации продолжается уже в процессе *A*. В последующем может произойти одно из двух: либо распространение ситуации будет прекращено в процессе *A*, либо процесс *A* будет прекращен. Если *A* в свою очередь подчинен некоторому процессу *C*, то распространение ситуации продолжится в *C*.

В случае, если независимый процесс *A* использует переменные, константы и объекты, локализованные в блоках другого процесса, необходимо учитывать следующее обстоятельство. Окончание выполнения этих блоков и ликвидация соответствующих данных может произойти еще до окончания процесса *A*. Последующее использование в *A* уничтоженных данных приведет к ошибке. Этого можно избежать, устроив соответствующим образом синхронизацию процессов.

Процесс создается с помощью стандартной процедуры *создпроцесс*, имеющей следующие параметры:

создпроцесс (*класс*, *p*, *x*1, ..., *xn*)

где *класс* — параметр, задающий класс процесса (0 — независимый, 1 — подчиненный); значением второго параметра является головная процедура процесса; *x*1, ..., *xn* — это параметры, передаваемые в головную процедуру. Параметры могут передаваться значением или именем *) (см. п. 11.1). Критическим блоком для подчиненного процесса является ближайший базированный блок, охватывающий место его создания.

Синхронизация процессов. Синхронизация осуществляется с помощью объекта типа семафор. Семафор может находиться в состоянии «открыт» или в состоянии «закрыт». Процесс может быть приостановлен на закрытом семафоре. Впоследствии процесс может быть возобновлен.

Существуют две «заготовки» для семафоров. Один семафор находится в открытом состоянии, а другой — в закрытом. Первый семафор является значением стандартной константы *семоткр*, а второй — значением стандартной константы *семзакр*. Для работы с семафором надо одну из заготовок присвоить переменной, и затем эту переменную передавать по имени в операции над семафором. При этом состояние семафора, являющегося значением переменной, будет соответствующим образом меняться. Ниже приводятся операции над семафорами. Идентификатором *c* обозначен параметр, имеющий синтаксическую форму ⟨переменная⟩. Параметр может быть передан не только «именем», но и «значением». Во втором случае следует передавать указатель переменной, содержащей семафор.

---

*) При передаче по имени, а также при передаче составных объектов (оперативных или внешних) надо учитывать, что выполнение блока, в котором локализован передаваемый объект или переменная, может закончиться раньше, чем выполнение процесса.

закрытьсем (имя *с*). Если семафор открыт, то операция переводит его в закрытое состояние, и процесс продолжается. В противном случае процесс приостанавливается на данном семафоре.

ждать (имя *с*). Если семафор открыт, то процесс продолжается, не закрывая его. В противном случае процесс приостанавливается на данном семафоре.

открытьсем (имя *с*). Если семафор открыт, то его состояние не изменяется. В противном случае семафор открывается и возобновляются все процессы, приостановленные на данном семафоре.

Если процессы были приостановлены в результате выполнения имп операции *закрытьсем*, то каждый из них прежде всего повторяет эту операцию.

Независимо от исходного состояния семафора процесс, выполнивший операцию *открытьсем*, продолжается.

пропустить (имя *с*). Отличие этой операции от предыдущей состоит в том, что: (а) семафор не открывается, и (б) каждый из возобновленных процессов, не повторяя операцию закрытия, начинает выполнять следующие за ней действия.

11.3. Задача. Задача характеризуется следующими особенностями:

— задача выполняется в собственной математической памяти;

— задача является той единицей вычислений, на которую начисляется расход ресурсов (объем используемой оперативной и вторичной памяти, время решения, время обмена и пр.);

— задача имеет ряд атрибутов, характеризующих текущий расход и лимит ресурсов, имя задачи и пр. (см. атрибуты задачи).

Конструкция ⟨запуск задачи⟩ может быть использована в двух целях:

— для запуска новой задачи в процессе выполнения процедуры (т. е. для запуска одной задачи из другой);

— для оформления пакетного задания (см. § 15 гл. 3).

⟨запуск задачи⟩ ::=

задача (⟨список установок атрибутов⟩)

⟨текст программы⟩{(⟨список фактических⟩)}

В результате выполнения ⟨запуска задачи⟩ создается новая задача. При необходимости, используя ⟨список установок атрибутов⟩, можно задать значения атрибутов этой задачи.

В рамках задачи создается независимый процесс, в котором роль головной процедуры играет заданная программа. Этот процесс может в свою очередь создавать процессы и задачи. ⟨Список фактических⟩ содержит параметры, передаваемые в головную процедуру процесса.

Задача ликвидируется, если (а) закончены все созданные в ней независимые процессы, и (б) не осталось прикрепленных к ней ацд- или им-файлов, для которых задана процедура реакции на активность со стороны терминала (см. п. 8.7 гл. 3). *

12. Программа

В этом параграфе вводится конструкция, определяющая общую синтаксическую структуру текста программы.

В простейшем случае текст программы это ⟨текст процедуры⟩ или ⟨закрытый оператор⟩, причем последний рассматривается как сокращенная запись процедуры без параметров:

проц ( ) ⟨закрытый оператор⟩

Ниже разобраны некоторые частные применения конструкций языка, позволяющих программировать простые случаи вызова программ из пакетных заданий. Для более подробного знакомства с возможностями языка в части обработки программ см. в гл. 3: § 1, пп. 2.1, 5.4 и §§ 12—15.

Программа задания может быть представлена в виде обычной программы автокода. Ниже приведен пример программы задания. В примере использовано то свойство, что текст одной программы может содержать текст другой программы. Последний обозначается с помощью конструкции ⟨изображение текстового файла⟩ (см. § 13, гл. 3).

Пример. Программа задания

```
начало
    ...
    тфайл (имяяз : ''авт'')
    *!!*
    текст
    проблемной
    программы
    !!;
    ...
конец
```

Здесь в тексте программы задания содержится ⟨изображение текстового файла⟩, начинающееся служебным словом **тфайл**. Между ограничителями *!!* и !! заключен текст вложенной программы. Значением данной конструкции является указатель (внешнего объекта), приводящий к изображенному текстовому файлу.

Вызов программы, как и вызов процедуры, осуществляется с помощью конструкции ⟨вызов⟩. Один из возможных типов значения ⟨первичного⟩ — это указатель файла текста программы. В результате происходит вызов процедуры, изо-

браженной в тексте программы *). Фактические параметры передаются обычным образом.

Пример. Вызов программы из задания

```
начало
    ...
  тфайл
  (имяяз : ''авт'')
  *!!*
  проц (...)
  начало
      ...
  конец
  !!
  (... фактические ...)
конец
```

Этот же пример можно запрограммировать иначе:

```
начало
  ...
  конст мояпрог = % описание константы
      тфайл (имяяз : ''авт'')
      *!!*
      проц (...)
      начало
          ...
      конец
      !! ;
  ..
  мояпрог (... фактические ...) ;
  ...
конец
```

Стандартный контекст. Считается, что глобально по отношению к программе описан ряд стандартных констант, значениями которых являются стандартные процедуры, стандартные ситуации и пр. Идентификаторы стандартных констант перечислены в Приложении 6.

---

*) В этом случае система осуществляет неявную трансляцию текста программы. Поэтому среди атрибутов текстового файла, т. е. в скобках после слова тфайл, надо задавать атрибут *имяяз*, определяющий внешнее имя транслятора используемого языка программирования. Отметим, что средствами языка можно явным образом задать такие действия, как вызов транслятора, «запись» результирующей программы в архив и пр.

⟨текст программы⟩: : —

программа {⟨описание⟩ ; } ... {⟨оператор⟩} ...

. . .⟨выражение⟩ **конец**

| ⟨изображение⟩

| ⟨закрытый оператор⟩

* *Общий случай.* Программа — это псевдообъект, содержащийся в файле объектного кода. Файл объектного кода представляет собой объект типа файл с соответствующим значением атрибута *типфайла.* Этот объект образуется в результате трансляции ⟨текста программы⟩.

⟨Текст программы⟩ в общем случае состоит из двух частей:

— первая часть — это последовательность ⟨описаний⟩ идентификаторов, составляющих собственный контекст программы. Считается, что идентификаторы стандартного контекста описаны глобально по отношению к собственному контексту;

— вторая часть задает действия, выполняемые при открытии программы.

Программу можно о т к р ы т ь и выполнить. Открытие осуществляется либо неявным образом, либо явно (см. п. 2.1 гл. 3). При открытии происходит следующее:

— выполняется инициализация программы, т. е. последовательность ⟨операторов⟩ и заключительное ⟨выражение⟩;

— значение ⟨выражения⟩ выдается в качестве результата открытия.

Результатом открытия может быть объект любого типа. Тип определяет последующее использование этого значения в открывающей программе.

В простейшем случае последовательность ⟨операторов⟩ пуста, а ⟨выражение⟩ — это ⟨текст процедуры⟩ или идентификатор некоторой процедуры из собственного контекста программы. Тогда результатом открытия является данная процедура. Используя ее в другой программе в конструкции ⟨вызов⟩, можно выполнить первую программу, как обычную процедуру.

В ы з о в п р о г р а м м ы. Значением ⟨первичного⟩ в конструкции ⟨вызов⟩ может быть не только процедура, но и указатель, приводящий к программе или к файлу объектного кода программы или к файлу текста программы. В этих случаях действия, связанные с открытием программы (и, если необходимо,— трансляция текста), производятся неявно. Далее полученная процедура выполняется обычным образом. Например, если значение переменной или константы ϕ приводит к файлу объектного кода или текста некоторой программы, то вызов этой программы с некоторыми параметрами x, y можно записать так: $\phi(x, y)$.

Аналогичным образом программа может быть использована в качестве головной процедуры параллельного процесса (см. п. 11.2).

С о к р а щ е н и я. Текст программы в виде ⟨изображения⟩ и ⟨закрытого оператора⟩ представляет собой сокращение соответственно для следующих случаев:

программа ⟨изображение⟩ **конец**

программа проц ( ) ⟨закрытый оператор⟩ **конец**

Собственный контекст может содержать только идентификаторы констант, но не идентификаторы переменных. Дополнительно накладываются следующие ограничения:

константа не может являться статической ситуацией;

выполнение ⟨описания⟩ не должно приводить к созданию переменных оперативных и внешних объектов (т. е. массивов и внешних объектов, отличных от константных).

Формально собственный контекст создается в неопределенный момент времени, до первого открытия программы, и затем переиспользуется последующими вызовами данной программы. Именно этим вызваны ограничения номенклатуры возможных ⟨описаний⟩.

В собственном контексте целесообразно сосредоточивать описания глобальных констант (чисел, процедур, текстовых макросов, полей), а также описания, в правой части которых запрограммировано обращение к архивным объектам по ⟨внешним именам⟩ (см. п. 5.4 гл. 3). В последнем случае знакомство с объектом (т. е. поиск его в архиве) происходит один раз на все открытия программы. *

13. Форматный обмен

В этом параграфе описывается один из существующих в языке способов обмена — форматный обмен. Особенность форматного обмена состоит в том, что он осуществляет: (а) вывод таких объектов, как числа и наборы, с преобразованием их из внутреннего представления в литерное и (б) ввод с обратным преобразованием из литерного представления во внутреннее. Остальные способы осуществляет двоичный обмен, при котором такое преобразование не производится (см. §§ 7 — 11 гл. 3).

Форматный обмен осуществляется с текстовым файлом. Последний представляет собой один из возможных вариантов внутренней организации файла. Структура текстового файла описана в § 12 гл. 3. Здесь приведены краткие предварительные сведения.

Текстовый файл представляет собой последовательность строк, состоящих из элементов литерного формата. Различаются файлы строк фиксированной длины и файлы строк плавающей длины. В первом случае длина всех строк файла одинакова и равна значению атрибута *длинстрок*. Во втором случае длина строки может изменяться от 0 до *махдлинблока*. Все строки файла могут быть пронумерованы или нет — в зависимости от значения атрибута *длинфикс*. Каждая строка нумерованного файла содержит собственный фиксированный номер строки.

Текущая позиция в файле — это позиция текущей литеры в текущей строке. Для отслеживания текущей позиции использу-

ется оперативный объект связи с файлом, имеющий тип позиционная переменная. Для случая форматного обмена существенны следующие атрибуты этого объекта: *фнс* — фиксированный номер текущей строки, *шагфнс* — шаг нумерации. Эти атрибуты определены только для нумерованных файлов.

Перед началом форматного обмена с некоторым файлом необходимо создать позиционную переменную. Для этого используется конструкция ⟨генератор объекта связи⟩ (см. п. 2.1 гл. 3). Например, пусть значение простой переменной (константы, формального параметра) *ф* приводит к файлу *). Тогда в результате выполнения оператора

позпф := *позп*(*ф*)

переменной *позпф* присваивается указатель позиционной переменной, необходимой для форматного обмена с файлом.

Другой способ использует то свойство, что параметром генерации позиционной переменной может быть не только ранее созданный файл, но и одна из стандартных констант, обозначающих тип внешнего устройства (ацпу, барабан, вывод на перфокарты и т. д.). В этом случае одновременно создается и файл и позиционная переменная. Например, пусть дана переменная *позпеч*, тогда в результате выполнения оператора

позпеч := *позп*(*ацпу*)

значением переменной *позпеч* станет указатель позиционной переменной вновь созданного файла на алфавитно-цифровом печатающем устройстве.

Первоначально текущая позиция установлена на начало файла. Используя операции позиционирования (см. п. 10.1 гл. 3), ее можно явным образом перемещать на конец файла или возвращать на начало.

Позиционная переменная является первым параметром операций форматного обмена. В процессе обмена происходит перемещение текущей позиции. При выводе последовательность литер, получившаяся в результате преобразования, помещается в текущую строку, начиная с позиции текущего элемента. При вводе последовательность литер, начинающаяся с текущей позиции, преобразуется в значение **) и присваивается заданной

*) Здесь подразумевается, что значение *ф* — это указатель оперативного объекта, связанного с файлом.

**) Здесь и далее в этом параграфе под термином «значение» понимается объект, полученный после преобразования при вводе, либо выводимый объект (до преобразования в литерное представление).

переменной. В обоих случаях текущая позиция сдвигается на элемент, следующий за последним обработанным. Существуют средства перехода на следующую строку и средства явной установки позиции текущего элемента строки.

Обмен с массивом. Форматный обмен можно осуществлять не только с файлом, но и с массивом, а именно со смежным вектором элементов литерного формата. Вектор рассматривается как файл, состоящий из одной строки. Роль позиционной переменной в данном случае выполняет указатель с-вектора. Операция обмена обрабатывает некоторое количество элементов и изменяет указатель таким образом, чтобы он описывал необработанную часть вектора. В связи с этим будем говорить, что операция продвигает указатель вперед на число обработанных литер.

Управление обменом. Существуют две разновидности форматного обмена: обмен, управляемый форматом, и обмен, управляемый данными.

В случае управления форматом преобразование осуществляется с помощью ⟨формата обмена⟩, который явным образом задается в операции. Формат описывает структуру и размер последовательности литер и тем самым задает действия, связанные с преобразованием «значение ↔ последовательность литер».

В случае управления данными обмен осуществляется с помощью одного из стандартных форматов обмена. Формат явно не указывается в операции, но выбирается по умолчанию. При выводе выбор определяется типом и форматом выводимого значения. При вводе выбор зависит от (а) структуры вводимой последовательности литер и (б) формата переменной, которой необходимо присвоить введенное значение.

Ошибки. В описании семантики указаны обстоятельства, приводящие к возникновению ошибок. Стандартная реакция системы на ошибки описана в п. 13.5.

13.1. Операции форматного обмена. В языке введены четыре операции форматного обмена: *запф* и *читф* — для обмена с файлом, и *запфм и читфм* — для обмена с массивом.

⟨форматный обмен⟩ ::=

запф(⟨выражение⟩, ⟨список элементов вывода⟩)

| *читф*(⟨выражение⟩, ⟨список элементов ввода⟩)

| *запфм*(⟨массив обмена⟩, ⟨список элементов вывода⟩)

| *читфм*(⟨массив обмена⟩, ⟨список элементов ввода⟩)

⟨элемент вывода⟩ ::=

⟨выражение⟩{: ⟨формат обмена⟩} | : ⟨позиционирование⟩

⟨элемент ввода⟩ ::=

имя переменная⟩ {: ⟨формат обмена⟩}
| ⟨выражение⟩{: ⟨формат обмена⟩}
| : ⟨позиционирование⟩
⟨массив обмена⟩ ::= ⟨выражение⟩ | имя ⟨переменная⟩

Первым параметром операций *запф* и *читф* является указатель позиционной переменной. Первым параметром операций *запфм* и *читфм* является указатель с-вектора. Этот параметр может быть передан значением или именем (см. п. 11.1). В случае передачи именем по окончании выполнения операции значением переданной переменной становится продвинутый указатель, описывающий часть вектора, начинающуюся с нового текущего элемента, т. е. необработанную часть вектора.

⟨Список элементов вывода⟩ и ⟨список элементов ввода⟩ может содержать элементы обмена, управляемого форматом, и (или) элементы обмена, управляемого данными. Элементы первой разновидности содержат компоненту «:⟨формат обмена⟩». В элементах второй разновидности такой компоненты нет.

⟨Выражение⟩, входящее в состав ⟨элемента вывода⟩, может задавать выводимое значение или массив выводимых значений. В первом случае значение ⟨выражения⟩ отлично от указателя массива. Во втором случае оно является указателем массива (с-вектора или в-массива). При выводе, управляемом форматом, значения всех элементов массива выводятся под управлением одного и того же формата, причем обработка каждого элемента осуществляется по правилам обработки одиночного значения.

⟨Переменная⟩, входящая в состав ⟨элемента ввода⟩, задает переменную, которой присваивается вводимое значение. ⟨Выражение⟩ задает массив, элементам которого присваиваются вводимые значения. Как и в случае вывода, обработка каждого элемента массива происходит по правилам обработки одиночной переменной.

Элемент обмена вида «:⟨позиционирование⟩» осуществляет перемещение текущей позиции, в частности, переход на следующую строку. Такой элемент не производит ввод или вывод какого-либо значения. Здесь, однако, можно в явном виде задать последовательность литер выводимой строки.

Элементы обмена выполняются слева направо в порядке текстуального расположения. ⟨Выражения⟩ и ⟨переменные⟩ выполняются один раз перед началом операции. Элементы n-мерного массива *) обрабатываются в следующем порядке: сначала

*) Если массив получен с помощью ⟨преобразования формы⟩ (см. § 15), то он должен быть прямоугольным параллелепипедом.

последовательно обрабатываются элементы с индексами $l_1, l_2, \ldots, l_{n-1}, l_n$, где $l_1, l_2, \ldots, l_{n-1}$ — нижние границы по соответствующим измерениям, а l_n пробегает весь допустимый диапазон значений индекса по n-му измерению, начиная от нижней границы и заканчивая верхней; затем индекс по измерению $n-1$ увеличивается на единицу и процесс повторяется и т. д. Таким способом осуществляется обход всех элементов массива. Если при выводе оказывается, что значением какого-либо элемента массива в свою очередь является указатель массива, то последний массив проходит такой же процесс обработки, а затем продолжается обработка первого. Исключение составляет случай вывода по ⟨трафарету тегированного⟩ (см. п. 13.4).

13.2. Обмен, управляемый данными.

Вывод. Обычно вывод начинается с текущей позиции. Если выводимое число не умещается на текущей строке, то предварительно выполняется переход на следующую строку. Если число помещается не в начало строки, то предварительно выводится один разделяющий пробел.

Тип целое. Выводимое число преобразуется в последовательность литер, размер которой в зависимости от формата числа равен: для ф32 — 11 литер (знак + 10 цифр), для ф64 — 20 литер (знак + 19 цифр); начальные нули заменяются на пробелы и знак помещается перед первой значащей цифрой.

Тип вещественное. Выводимое число преобразуется в последовательность литер, имеющую вид TU.V*e*TP, где T — знак числа или порядка, U — отличная от нуля цифра целой части мантиссы, V — дробная часть мантиссы, *e* — символ порядка, P — порядок. Размер дробной части и порядка зависит от формата числа:

ф32: V— 7 цифр, P— 2 цифры;

ф64: V— 16 цифр, P— 2 цифры;

ф128: V— 33 цифры, P— 5 цифр.

Начальные нули порядка заменяются на пробелы, знак порядка помещается перед первой значащей цифрой.

Тип набор. Набор преобразуется в целое формата ф64 и выводится как целое число формата ф64 без знака. В случае, если старший знаковый разряд исходного полного набора (т. е. разряд, соответствующий знаку целого) равен 1, возникает «ошибка значения».

Массив. Если массив состоит из элементов простого формата, то значения элементов массива выводятся по правилам вывода чисел и наборов.

В случае, если массив состоит из элементов строчного формата, каждый элемент выводится как последовательность, состоящая из одной литеры. Литера представляет собой ⟨двоичную цифру⟩ для формата ф1, ⟨шестнадцатеричную цифру⟩ для формата ф64 и ⟨литеру⟩ для формата ф8. Разделяющий пробел перед литерой не выводится. При необходимости выполняется переход на следующую строку.

Попытка вывода значений других типов приводит к «ошибке значения».

Ввод. Размер и структура вводимой последовательности литер зависит от того, что собою представляет элемент ввода.

Переменная простого формата. Начиная с текущей позиции, отыскивается первая литера, отличная от пробела (при этом, если нужно, происходит переход на следующую строку). Далее набирается последовательность литер, подчиняющаяся синтаксису ⟨целого⟩ или ⟨вещественного⟩. Числу может предшествовать знак, возможно, отделенный от числа пробелами. Допускаются пробелы после знака порядка и символа порядка *е*. Это число преобразуется во внутреннее представление и присваивается переменной. Если величина числа такова, что оно не может быть преобразовано к формату переменной, возникает «ошибка значения». Если последовательность литер не подчиняется синтаксису ⟨целого⟩ либо ⟨вещественного⟩, возникает «ошибка литеры».

Переменная строчного и нестандартного формата. Тем же способом отыскивается целое число без знака, преобразуется во внутреннее представление и присваивается переменной. «Ошибка значения» возникает, если величина числа больше, чем $2^{\phi} - 1$, где ϕ — формат переменной.

Массив. Если массив состоит из элементов простого формата, то каждый его элемент обрабатывается как переменная простого формата. Если массив состоит из элементов строчного формата, то каждый его элемент обрабатывается следующим образом:

1. Форматы ф1 и ф4: начиная с текущей позиции, отыскивается первая литера, отличная от пробела (при необходимости выполняется переход на следующую строку). В случае формата ф1 литера должна быть ⟨двоичной цифрой⟩, а в случае формата ф4 — ⟨шестнадцатеричной цифрой⟩, иначе возникает «ошибка литеры». Литера преобразуется во внутреннее представление, соответствующее двоичным или шестнадцатеричным цифрам, и присваивается элементу массива.

2. Формат ф8: литера, находящаяся в текущей позиции, присваивается элементу массива. При этом, если в текущей строке

нет больше литер, то предварительно выполняется переход на следующую строку.

13.3. Позиционирование. ⟨Позиционирование⟩ обеспечивает следующие основные возможности: установку новой текущей позиции, вывод заданной последовательности литер.

⟨позиционирование⟩ ::= ⟨литерал⟩{⟨размещение⟩} ...
| ⟨размещение⟩ ...
⟨размещение⟩ ::= {⟨повторитель⟩}
⟨код размещения⟩{⟨литерал⟩}
⟨литерал⟩ ::= {⟨повторитель⟩"⟨литера⟩ ..."} ...
{⟨повторитель⟩}"⟨литера⟩..."

⟨код размещения⟩ ::= *x*|*y*|*к*|*l*|*p*
⟨повторитель⟩ ::= ⟨целое⟩|*n* (⟨выражение⟩)

В последнем правиле буква *n* — это латинское N.

Установка позиции задается с помощью компоненты ⟨размещение⟩. Конкретные действия обозначаются соответствующим ⟨кодом размещения⟩:

x — текущая позиция смещается вправо на один элемент. Если при выводе это приводит к появлению незаполненного элемента, то в него помещается пробел.

y — текущая позиция смещается влево на один элемент.

к — текущим элементом становится элемент строки с номером, равным значению ⟨целого⟩ или ⟨выражения⟩ из ⟨повторителя⟩. Если при выводе это приводит к появлению незаполненных элементов, то в них помещаются пробелы.

l — текущая позиция перемещается в начало следующей строки. При выводе имеют место следующие особенности. В случае файла строк фиксированной длины незавершенная часть выводимой строки заполняется пробелами. В случае файла строк плавающей длины выводится последовательность литер, заключенная между началом строки и текущей позицией. При выводе в нумерованный файл новой строке присваивается номер, равный сумме текущих значений атрибутов *фнс* и *шагфнс*. Это же число присваивается атрибуту *фнс*. В операциях обмена с массивом *запфм* и *читфм* подобный ⟨код размещения⟩ игнорируется.

p — происходит переход на первую строку следующей страницы. Такое действие имеет смысл только для ацпу-файлов. В остальных случаях подобный ⟨код размещения⟩ игнорируется.

Компонента ⟨литерал⟩ имеет следующий смысл. При выводе последовательность литер, заключенная между кавычками, выводится, начиная с текущей позиции, как массив литер (см.

и. 13.2). При вводе проверяется, совпадает ли последовательность литер, начинающаяся с текущей позиции, с последовательностью, заключенной между кавычками (в процессе сравнения, если надо, осуществляется переход на следующую строку). В случае несовпадения возникает «ошибка литеры».

Число повторений. ⟨Целое⟩ или значение ⟨выражения⟩ в ⟨повторителе⟩ задает число повторений соответствующего действия (операции позиционирования, операции вывода или сравнения ⟨литерала⟩). Если ⟨повторитель⟩ опущен, действие выполняется один раз. Исключение составляет случай ⟨кода размещения⟩ *, когда ⟨повторитель⟩ имеет другой смысл (см. выше) и должен присутствовать обязательно.

13.4. Обмен, управляемый форматом.

Вывод. Значение преобразуется в последовательность литер, структура которой определяется ⟨форматом обмена⟩. Каждый формат накладывает определенные ограничения на тип и величину значений, выводимых по данному формату. Если эти ограничения не выполняются, возникает «ошибка значения». Последовательность литер выводится, начиная с текущей позиции.

Ввод. Последовательность литер, начинающаяся с текущей позиции, проходит обратное преобразование. Структура вводимой последовательности должна соответствовать ⟨формату обмена⟩, иначе возникает «ошибка литеры». Значение, полученное в результате преобразования, присваивается заданной переменной. Формат последней накладывает ограничения на тип и величину вводимых значений. Если эти ограничения не удовлетворяются, возникает «ошибка значения».

Ввод/вывод массива. Если особо не оговорено, то при выводе массива значение каждого элемента массива выводится по правилам обработки одиночного значения. Аналогично при вводе каждый элемент массива обрабатывается как одиночная переменная. Обработка всех элементов происходит под управлением одного и того же ⟨формата обмена⟩.

⟨формат обмена⟩ ::= {⟨вставка⟩}⟨трафарет⟩
⟨трафарет⟩ ::=
⟨трафарет числа⟩ | ⟨трафарет целого⟩
| ⟨трафарет вещественного⟩ | ⟨трафарет литерного⟩
| ⟨трафарет логического⟩ | ⟨трафарет двоичного⟩
| ⟨трафарет тегированного⟩

Каждый конкретный ⟨трафарет⟩ управляет вводом/выводом последовательности литер, имеющей некоторую специфическую форму. Например, ⟨трафарет вещественного⟩ описывает после-

довательность, имеющую форму вещественного числа с плавающей точкой (экспоненциальная форма) или числа с фиксированной точкой.

По ⟨трафаретам⟩, связанным с вводом/выводом чисел, можно определить общее количество цифр в числе. Это количество ограничено: для целых — не более 19 цифр, а для вещественных — не более 34 цифр мантиссы и не более 5 цифр порядка.

В с т а в к а. При выводе ⟨вставка⟩ предписывает поместить в выводимую последовательность заданные литеры. При вводе проверяется, что заданные литеры содержатся во вводимой последовательности. Далее эти литеры из рассмотрения исключаются и не принимают участие в формировании внутреннего представления вводимого значения. Если результат проверки отрицательный, то возникает «ошибка литеры».

⟨вставка⟩ : =⟨литерал⟩{⟨смещение⟩}...|⟨смещение⟩...
⟨смещение⟩ ::= {⟨повторитель⟩} *x* {⟨литерал⟩}

Компонента ⟨смещение⟩ имеет такой же смысл, как ⟨размещение⟩ с ⟨кодом размещения⟩ *x*. ⟨Повторитель⟩ используется не только во ⟨вставках⟩, но и с тем же смыслом — внутри ⟨трафаретов⟩. Исключение составляет ⟨трафарет числа⟩, где ⟨повторитель⟩ используется в ином смысле (см. ниже).

* Значения ⟨выражений⟩ всех ⟨повторителей⟩ всех элементов обмена выполняются один раз перед началом обмена. На ⟨повторители⟩ ⟨литералов⟩, входящих в состав ⟨формата обмена⟩, наложено следующее ограничение: здесь допускаются только ⟨выражения⟩ статического класса. Это ограничение не распространяется на ⟨литералы⟩, входящие в состав ⟨позиционирования⟩. *

Трафарет числа. Трафарет числа управляет вводом/выводом последовательности литер в форме целого числа, числа с фиксированной точкой или числа с плавающей точкой. Он обеспечивает наиболее простой способ ввода/вывода чисел, являющийся промежуточным между обменом, управляемым данными, и обменом, управляемым форматом.

⟨трафарет числа⟩ ::=
{⟨повторитель⟩}*g*{(⟨выражение⟩{ , ⟨выражение⟩
{ , ⟨выражение⟩}})} {⟨вставка⟩}

В ы в о д. Форма выводимой последовательности определяется количеством ⟨выражений⟩ в скобках: 1 — целое число, 2 — число с фиксированной точкой, 3 — число с плавающей точкой. Если скобки и заключенные в них ⟨выражения⟩ опущены, то вывод производится по правилам обмена, управляемого данными. Обозначим значения ⟨выражений⟩: *ww*, *wf*, *we*.

Целое число. *ww* задает общий размер последовательности. Незначащие нули числа заменяются на пробелы. Знак помещается перед первой значащей цифрой. Отрицательный знак *ww* означает, что знак включается в общий размер и выводится только для отрицательных чисел. Если *ww* равно нулю, то выводится последовательность минимальной возможной длины (знак выводится только для отрицательных чисел). Если *ww* отлично от нуля и выводимое число не умещается в размер *ww*, то возникает «ошибка значения».

Пример.

По трафарету *g*(—4) можно вывести числа *):

␣␣␣0 ␣␣12 ␣—12 1234

по трафарету *g*(4) можно вывести

␣␣+0 ␣+12 ␣—12

но число 1234 вывести нельзя.

По трафарету *g*(0) можно вывести

0 12 123 1234

Число с фиксированной точкой. *ww* задает общий размер последовательности, *wf* — число цифр после десятичной точки. Знак *ww* и значение *ww*, равное нулю, трактуются так же, как и в предыдущем случае. Если величина числа такова, что перед десятичной точкой не умещаются все цифры целой части, то количество позиций, отводимых для дробной части, сокращается. Если это количество обратилось в нуль, но число тем не менее не умещается, возникает «ошибка значения».

Пример.

По трафарету *g*(—6,3) можно вывести
␣1.234 —1.234 12.345 123.45

1234.5 ␣12345 123456
В последних трех случаях число позиций после точки сократилось.

По трафарету *g*(0, 3) можно вывести

1.234 —1.234 123.456

Число с плавающей точкой. *ww* задает общий размер последовательности, *wf* — число цифр дробной части, *we* —

*) Символ ␣ означает пробел в выводимой последовательности.

число цифр порядка. Последовательность литер имеет форму TU.VeTP, где T — знак мантиссы и порядка, U — поле цифр целой части, V — поле цифр дробной части, *e* — символ порядка, P — поле цифр порядка.

Независимо от знака *ww* под знак мантиссы всегда отводится первая позиция. Если знак *ww* — отрицательный, то положительный знак мантиссы кодируется пробелом. Мантисса выравнивается относительно десятичной точки таким образом, чтобы следующая за знаком первая цифра целой части была ненулевая. Далее выводятся остальные цифры целой и дробной частей мантиссы. Порядок выводится по правилам вывода целого числа по трафарету *g*(*we*). Знак *we* трактуется так же, как описано выше. Если *we* равно нулю, то предпринимается попытка представить по возможности большее число значащих цифр числа. С этой целью размеры поля порядка и поля целой части выбираются так, чтобы первое было минимально необходимым, а второе — максимально возможным.

Если величина порядка такова, что он не умещается в поле порядка, число позиций, отводимых для мантиссы, сокращается; сначала сокращается число позиций поля дробной части, а затем — число позиций поля целой части. При этом, однако, требуется, чтобы осталась по крайней мере одна цифра целой части, иначе возникает «ошибка значения».

Размер выводимой последовательности всегда равен *ww*, которое в рассматриваемом случае должно быть отлично от 0.

Пример.

По трафарету *g*(10, 3, 3) можно вывести

$-1.234e\_+0$ $+1.234e-12$ $+1.23e+456$ $+1e\_+12345$

В двух последних случаях число позиций поля дробной части сократилось.

По трафарету *g*(6, 1, 0) можно вывести

$-123e4$ $+12e34$ $+11e-3$

В последнем случае подразумевается, что выводимое число равно, например, 0.011.

По ⟨трафарету числа⟩ можно выводить целые и вещественные числа. Вывод вещественного числа по трафарету *g*(*ww*) эквивалентен выводу по трафарету *g*(*ww*, 0).

Массив. Вывод массива производится поэлементно под управлением одного и того же трафарета.

⟨Повторитель⟩ позволяет управлять количеством чисел, располагающихся на одной строке файла. Пусть значение ⟨повторителя⟩ равно *k*. Тогда в строке, начиная с левой крайней

позиции, равномерно размещается k чисел; каждое число занимает ww позиций. ww в данном случае должно быть отлично от нуля. Каждая строка массива выводится с новой строки файла. Под строкой массива здесь понимается совокупность элементов n-мерного массива, получаемая путем фиксации первых $n-1$ индексов.

Перед выводом начальной строки массива выводится ⟨вставка⟩, если она есть; затем проверяется, находится ли текущая позиция в начале строки. Если нет — осуществляется переход на новую строку. После вывода последней строки всегда происходит переход на новую строку.

Если выводится не массив, а одиночное значение, то ⟨повторитель⟩ игнорируется.

В в о д. При вводе по ⟨трафарету числа⟩ все компоненты трафарета, кроме ⟨вставки⟩, игнорируются, и вывод происходит по правилам обмена, управляемого данными. ⟨Вставка⟩, если она есть, обрабатывается обычным образом.

* *Трафарет целого.* ⟨Трафарет целого⟩ управляет вводом/выводом последовательности литер, имеющей форму целого числа со знаком или без знака.

⟨трафарет целого⟩ ::= {⟨поле знака⟩}⟨поле цифр⟩
⟨поле знака⟩ ::=
⟨знак⟩{⟨вставка⟩}⟨поле плавающего знака⟩
{⟨вставка⟩}
⟨поле цифр⟩ ::= ⟨подполе цифр⟩...
⟨подполе цифр⟩ ::=
{⟨повторитель⟩}{*s*} *d* {⟨вставка⟩
| ⟨повторитель⟩} *z* {⟨условная вставка⟩}
| {⟨повторитель⟩}*z* ⟨вставка⟩
⟨поле плавающего знака⟩ ::=
⟨подполе плавающего знака⟩...⟨знак⟩
подполе плавающего знака⟩ ::=
{⟨повторитель⟩} *f* {⟨условная вставка⟩}
{⟨повторитель⟩} *f* ⟨вставка⟩
⟨условная вставка⟩ ::= *c* ⟨вставка⟩

В трафарете возможны следующие редакционные особенности:

1. Замена незначащих и значащих нулей на пробелы (с помощью компоненты ⟨подполе цифр⟩ с *z*-спецификацией).
2. Замена незначащих нулей на пробелы и вставка знака перед первой значащей цифрой (с помощью компоненты ⟨поле плавающего знака⟩).
3. Редакционные вставки (с помощью компоненты ⟨вставка⟩ и ⟨условная вставка⟩).

4. Отбрасывание некоторых цифр числа (с помощью компоненты ⟨подполе цифр⟩ с *s*-спецификацией).

Для целого без знака (в трафарете опущено ⟨поле знака⟩) в случае, если в трафарете не заданы редакционные вставки и отбрасывание цифр, соответствующая последовательность литер представляет собой последовательность цифровых позиций (поле цифр)

yy . . . y

где *y* — пробел или цифра.

В простейшем случае ⟨трафарет целого⟩ состоит из одного ⟨подполя цифр⟩ с *d*-спецификацией или одного ⟨подполя цифр⟩ с *z*-спецификацией. Размер вводимого или выводимого поля цифр задается ⟨повторителем⟩.

d-спецификация задает безусловное подполе цифр. Все позиции безусловного подполя заполнены цифрами числа. Незначащие нули, если таковые есть, представлены символом 0. Например, по трафарету 5*d* число 2 представляется в форме 00002.

z-спецификация задает условное подполе цифр. При выводе начальные нули условного подполя (незначащая часть подполя) заменяются на пробелы. Во вводимой строке начальные нули могут быть представлены нулями или пробелами. Значащая часть подполя, т. е. часть, начинающаяся с первой значащей цифры, заполнена цифрами числа. Например, по трафарету 5*z* число 2 представляется в форме *) ␣␣␣␣2.

Поле цифр в общем случае состоит из безусловных и условных подполей. Размер поля цифр равен сумме размеров составляющих его подполей ($wd + wz$, где wd — суммарный размер безусловных подполей, wz — суммарный размер условных подполей). Позиции подполей заполняются по указанным выше правилам. В частности, начальные нули условного подполя при выводе заменяются на пробелы независимо от того, является ли оно начальным подполем поля цифр или каким-либо внутренним подполем. Следует учитывать, что последовательность условных подполей (возможно, включающая редакционные вставки) образует единое условное подполе.

Например, по трафарету 3*z*2*d* число 21012 представляется в форме 21012, а число 1012 — формой ␣1012, число 12 — формой ␣␣␣12, число 2 — формой ␣␣␣02, число 0 — формой ␣␣␣00. Трафареты 2*zz*2*d* и *zzz*2*d* эквивалентны 3*z*2*d*.

Вставки. Различаются безусловные и условные редакционные вставки. Безусловная редакционная вставка задается компонентой ⟨вставка⟩ (семантика ⟨вставки⟩ описана выше). Например, по трафарету "x = "4*zd* число 12 представляется формой x = ␣␣␣12, по трафарету *zd*" — "*zd*" — "3*zd* число 6051980 представляется формой ␣6—␣5—1980.

В условном подполе цифр может быть задана условная редакционная вставка. Литеры вставки помещаются в соответст-

*) Здесь и далее символ ␣ означает пробел в выводимой или вводимой последовательности литер.

вующие позиции подполя только в том случае, если вставка попадает в значащую часть подполя. Если же вставка попадает в незначащую часть подполя, соответствующие ей позиции заполнены проблемами. Например, с помощью трафарета *3zc*", *3zc*", "*2zd* можно разделить запятой разряда числа, соответствующие миллионам, тысячам и сотням.

Пример.

1. Число 100000000 представляется формой 100, 000, 000,
2. Число 10000000 представляется формой ␣␣1, 000, 000.
3. Число 1000 представляется формой ␣␣␣␣␣1,000

(первая запятая заменена на пробел).

При вводе требуется, чтобы в позициях, соответствующих условной вставке, находились литеры вставки или пробелы.

Отбрасывание цифр. С помощью *s*-спецификации можно задать отбрасывание всех цифр какого-либо безусловного подполя цифр. Это означает, что при выводе цифры соответствующего подполя числа отбрасываются и не появляются в выводимой последовательности. При вводе данное подполе вставляется во вводимую последовательность. Позиции вставляемого подполя заполнены нулями. Например, с помощью трафарета *2zd6sd* можно задать масштабирование с масштабом 6 десятичных позиций. При этом число 1000000 будет выводится в форме ␣␣1, а последовательность 123 преобразовывается при вводе в число 123000000.

Целое с фиксированным знаком. Если ⟨поле знака⟩ представлено ⟨знаком⟩ (а не ⟨полем плавающего знака⟩), то такой трафарет задает последовательность в форме целого со знаком, находящимся в фиксированной позиции. Если не учитывать возможные редакционные вставки и отбрасывание цифр, то размер последовательности равен $wd + wz + 1$. Литера, представляющая знак, находится в начальной позиции. Например, число 12 по трафарету *+4zd* представляется в форме +␣␣␣12.

Заполнение поля цифр определяется в зависимости от трафарета таким же образом, как и для целого без знака.

Если ⟨знак⟩ есть + (−), то для положительных чисел в знаковой позиции находится литера + (пробел), а для отрицательных — литера − (−). Например, число −2 по трафарету "$x =$" *+4zd* представляется в форме $x =$ −␣␣␣2, а число 2 — в форме $x =$ ␣␣␣␣2.

Целое с плавающим знаком. Наличие в трафарете компоненты ⟨поле плавающего знака⟩, как и в предыдущем случае, задает последовательность литер в форме целого со знаком. Однако здесь знак помещается не в фиксированную позицию, а в одну из позиций поля знака. Размер этого поля равен $we + 1$, где we — сумма размеров подполей плавающего знака. В позициях этого поля незначащие нули заменены на пробелы (при вводе и при выводе), а символ, представляющий знак, помещен перед первой значащей цифрой. Например, число −2 по трафарету "$x =$ "*4f* + *d* представляется в форме $x =$ ␣␣␣−2, а число −102 — в форме $x =$ ␣␣−102.

Если первая значащая цифра находится вне поля знака, то знак помещается в последнюю позицию поля. Например, число −2 по трафарету "$x =$ "*3f* + *2d* представляется в форме $x =$ ␣␣␣−02.

Заполнение позиций поля цифр определяется в зависимости от трафарета таким же образом, как и для целого без знака. Например, число —2 по трафарету "$x = "3f + zd$ представляется в форме $x =$ ␣␣␣—␣2.

Компонента ⟨знак⟩ употребляется в ⟨поле плавающего знака⟩ с тем же смыслом, что и для случая фиксированного знака.

Внутри ⟨поля плавающего знака⟩ может быть задана ⟨условная вставка⟩. Если первая значащая цифра встречается до позиции, с которой начинается вставка, то символы вставки помещаются в строку. В противном случае вместо вставки в строку помещается число пробелов, равное размеру вставки. Если первая значащая цифра непосредственно следует за вставкой, то вставка также заменяется пробелами, но в последнюю позицию поля вставки помещен символ, представляющий знак числа, например, число 100000 по трафарету $4fc","f + d$ представляется в форме +100,000, число —1000— в форме ␣␣—1,000, а число 100 — в форме ␣␣␣␣+100.

Заполнение поля цифр определяется в зависимости от трафарета таким же образом, как и для целого без знака. Например число —2 по трафарету $4fc","+ zd$ представляется в форме ␣␣␣␣␣—␣2.

Общий размер вводимой/выводимой последовательности литер равен $wt + wd + wz + wi + wc$, где wt — размер поля знака, равный 0 (если поле знака отсутствует), или 1 (в случае фиксированного знака), или $wf + 1$ (в случае плавающего знака); wi — суммарный размер безусловных вставок; wc — суммарный размер условных вставок.

Число значащих цифр выводимого целого не должно превышать $wd + wz$. По ⟨трафарету целого⟩ без знака можно выводить только неотрицательные числа. При вводе допускаются переменные стандартного и нестандартного формата. В случае строчного и нестандартного формата можно вводить только неотрицательные числа. Зависимость максимальной абсолютной величины вводимого числа от формата переменной такая же, как и в случае ввода управляемого данными.

Трафарет вещественного. ⟨Трафарет вещественного⟩ предназначен для ввода/вывода последовательности литер, имеющей одну из трех форм вещественного числа:

TU.V (число с фиксированной точкой)
TU.V*e*TP (число с плавающей точкой)
TU*e*TP (число с плавающей точкой)

Здесь T — поле знака (которое может отсутствовать), U — поле цифр целой части, V — поле цифр дробной части, *e* — символ порядка, P — поле цифр порядка.

По ⟨трафарету вещественного⟩ можно выводить вещественные и целые числа. При вводе допускаются только переменные простого формата. Абсолютная величина вводимого числа должна быть такова, чтобы его можно было преобразовать к формату переменной, иначе возникает «ошибка значения».

⟨трафарет вещественного⟩ ::=
⟨трафарет с фиксированной точкой⟩
| ⟨трафарет с плавающей точкой⟩

⟨трафарет с фиксированной точкой⟩ ::=
, ⟨трафарет целого⟩ ⟨поле точки⟩ ⟨поле цифр⟩

⟨трафарет с плавающей точкой⟩ ::=
⟨трафарет с фиксированной точкой⟩⟨поле порядка⟩
| ⟨трафарет целого⟩⟨поле порядка⟩

⟨поле точки⟩ ::= {*s*}. {⟨вставка⟩}

⟨поле порядка⟩ ::= {*s*}*e* ⟨трафарет целого⟩

Число с фиксированной точкой. Компонента ⟨трафарет целого⟩ предназначена для ввода/вывода знака и цифры целой части числа. Соответствующие позиции обрабатываются по правилам, описанным в разделе трафарет целого.

Компонента ⟨поле цифр⟩ предназначена для ввода/вывода цифр дробной части числа. Позиции этой части заполняются по правилам, описанным для трафарета целого в случае целого без знака. Единственное отличие состоит в том, что если поле цифр целой части заканчивается подполем с *z*-спецификацией, а поле цифр дробной части начинается с такого подполя, то эти подполя объединяются в смысле замены незначащих нулей на пробелы. Т. е. если первая значащая цифра встретилась в подполе целой части, то после нее (в частности, в подполе дробной части) помещаются цифры числа. Если первая значащая цифра встретилась в подполе дробной части, то незначащие нули в целой и дробной части заменяются на пробелы. Если же поле цифр целой части заканчивается подполем с *d*-спецификацией, то поле цифр дробной части рассматривается как независимое целое число.

Компонента ⟨поле точки⟩, во-первых, задает позицию десятичной точки для выравнивания числа при выводе, во-вторых, определяет, каким символом разделены целая и дробная часть. Если в ⟨поле точки⟩ опущена *s*-спецификация, то число представлено в обычном виде, т. е. целая и дробная часть разделены символом точки. Например, число 123.456 по трафарету $3f + d.3d$ представляется в форме +123.456, а число 0.123 — в форме ␣␣␣+0.123.

Если в ⟨поле точки⟩ указана *s*-спецификация, то оно только задает положение подразумеваемой десятичной точки для выравнивания. Символ точки в строку не помещается. Например, с помощью трафарета "x = "*sds.3sd3d* можно задать масштабирование. Число 0.000123 по этому трафарету представляется в форме x = 123.

Трафарет *d*", "*s*.2*d* позволяет десятичную точку представить литерой «, ». Например, число 3,14 представляется в форме 3,14.

Общий размер вводимой/выводимой последовательности равен $wt + wd + wz + wi + wc + 1 - ws$, где ws — число отбрасываемых литер (цифр и/или символа точки), а остальные обозначения те же, что использованы при описании ⟨трафарета целого⟩. Количество значащих цифр целой части выводимого числа не должно превышать размера поля цифр, задаваемого ⟨трафаретом целого⟩.

Число с плавающей точкой. Компоненты ⟨трафарет с фиксированной точкой⟩ и ⟨трафарет целого⟩ предназначены для ввода/вывода десятичной мантиссы числа с плавающей точкой. В случае ⟨трафарета целого⟩ подразумеваемая

десятичная точка находится справа от последней цифры мантиссы. При выводе число выравнивается относительно десятичной точки так, чтобы первая цифра целой части мантиссы была отлична от нуля. Порядок, получившийся в результате выравнивания, выводится в поле цифр порядка. При вводе не требуется, чтобы первая цифра мантиссы была ненулевая. Мантисса обрабатывается по правилам, описанным для числа с фиксированной точкой или для целого, и затем домножается на десять в степени, равной порядку.

Компонента ⟨трафарет целого⟩ в ⟨шаблоне порядка⟩ предназначена для ввода/вывода порядка числа. Эта часть строки обрабатывается как независимое целое число по правилам, описанным для ⟨трафарета целого⟩.

Наличие *s*-спецификации в ⟨шаблоне порядка⟩ означает, что символ порядка не отделяет поле мантиссы от поля порядка.

Например, число 123,456 по трафарету $d.2de+2d$ представляется строкой $1.23e+02$, а по трафарету $d.2dse+2d$ — строкой 1.23+02. Используя в ⟨шаблоне порядка⟩ комбинации из *s*-спецификации и ⟨вставки⟩, можно поле мантиссы отделить от поля порядка любым другим подходящим символом или последовательностью символов.

Общий размер вводимой/выводимой строки равен $wm+wp+wip+we$, где wm и wp — размер поля мантиссы и поля порядка, определяемый так же, как для чисел с фиксированной точкой и целых чисел; wip — размер вставки в ⟨поле порядка⟩; we равно 1, если *s*-спецификация в ⟨поле порядка⟩ опущена, и равно 0 в противном случае. Абсолютная величина порядка выводимого числа не должна превышать величину, допустимую для поля порядка. В противном случае возникает «ошибка значения».

Трафарет логического. ⟨Трафарет логического⟩ предназначен для ввода/вывода литеры, обозначающей логическое значение (*и* — **истина**, *л* — **ложь**).

⟨трафарет логического⟩ ::= *b* {⟨вставка⟩}

При выводе допускаются целые и наборы, числовая величина которых равна 0 (**ложь**) или 1 (**истина**). При вводе литера преобразуется в логическое значение (т. е. в одноэлементный битовый набор) и присваивается переменной. Допускаются переменные любых форматов.

Трафарет литерного. ⟨Трафарет литерного⟩ предназначен для ввода/вывода последовательности литер.

⟨трафарет литерного⟩ ::= ⟨подполе литер⟩ ...

⟨подполе литер⟩ ::= {⟨повторитель⟩}{*s*} *a* {⟨вставка⟩}

Трафарет *Ra*, где *R* — ⟨повторитель⟩, задает ввод или вывод очередных *R* литер. Трафарет *Rsa* задает пропуск очередных *R* литер выводимого значения; при вводе в результирующую последовательность вставляется *R* пробелов; в обоих случаях текущая позиция по файлу не сдвигается. ⟨Вставка⟩ обрабатывается обычным образом. В общем случае позиция по файлу сдвигается на $wi+wa$ литер, где wi — суммарный размер вставок, wa — суммарное число литер, обработанных по трафарету *Ra*. Если необходимо, осуществляется переход на следующую строку.

При выводе допускается набор (возможно, неполный) или смежный вектор элементов литерного формата. Число элементов в обоих случаях должно быть равно $wa + ws$, где ws — число литер, обрабатываемых по трафарету *Rsa*. Иначе возникает «ошибка значения». Литеры обрабатываются слева направо.

При вводе допускается переменная любого формата или с-вектор элементов литерного формата. В первом случае $wa + ws$ литер преобразуются в набор, и набор присваивается переменной. Во втором случае $wa + ws$ литер присваиваются соответствующим элементам вектора. Формат переменной должен быть не меньше $wa + ws$; длина вектора должна быть равна $wa + ws$, иначе возникает «ошибка значения».

Трафарет двоичного. Трафарет двоичного предназначен для ввода/вывода последовательности литер во внутреннем представлении.

⟨трафарет двоичного⟩ ::= ⟨основание⟩*r*⟨поле цифр⟩{⟨вставка⟩}

⟨основание⟩ ::= 1 | 3 | 4

Компонента ⟨поле цифр⟩ предназначена для ввода/вывода ВРП значения в двоичном, восьмеричном или шестнадцатеричном виде (в зависимости от ⟨основания⟩). Структура последовательности литер определяется, как и в случае целого без знака. В каждой цифровой позиции находится соответствующая цифра (двоичная, восьмеричная или шестнадцатеричная) ВРП значения. При выводе рассматриваются правые разряды выводимого значения. Число рассматриваемых разрядов равно $(wz + wd) * wr$, где wr равно 1, 3 или 4.

При вводе происходит обратное преобразование: последовательность литер преобразуется в набор (в общем случае — неполный) и присваивается переменной.

При выводе допускаются целые и вещественные числа и наборы. Массив обрабатывается поэлементно, как при выводе чисел. При вводе допускаются переменные и массивы элементов любых форматов.

Количество обрабатываемых разрядов $(wz + wd) * wr$ не должно превышать 64.

Трафарет тегированного. ⟨Трафарет тегированного⟩ предназначен для ввода/вывода последовательности литер, представляющей значение во внутреннем представлении с тегом.

⟨трафарет тегированного⟩ ::= ⟨основание⟩*t*{⟨вставка⟩}

Последовательность литер имеет форму $TT\_YY \ldots Y$, где TT — две шестнадцатеричные цифры 6-разрядного тега значения, $YY \ldots Y$ — цифры ВРП значения. В зависимости от основания длина этого поля равна 64 (двоичное представление), 22 (восьмеричное представление), или 16 (шестнадцатеричное представление) цифрам.

Последовательность литер, представляющая вещественное формата ф128, состоит из двух таких последовательностей, разделенных пробелом.

При выводе допускаются значения любого типа. В частности, указатель массива трактуется как выводимое значение.

При вводе допускаются переменные и массивы элементов любых форматов. Вводимая последовательность литер преобразуется в значение, тип которого определяется полем TT и

присваивается переменной или элементу массива. Подобным образом нельзя вводить адресные значения (указатель, метка). В противном случае возникает «ошибка значения».

13.5. Ошибки форматного обмена. В этом пункте описываются стандартные реакции системы на ошибки форматного обмена.

Ошибка значения. При выводе возможны следующие случаи:

— встречено значение недопустимого типа или числовая величина значения такова, что оно не может быть выведено под управлением заданного формата. В этом случае выводится последовательность литер «*». При выводе под управлением формата длина последовательности определяется этим форматом; при выводе, управляемом данными, длина равна 20-ти литерам (т. е. размеру целого формата ф64);

— длина выводимого вектора (или набора) отличается от длины, заданной ⟨форматом литерного⟩. Если длина меньше заданной, то к выводимой последовательности справа добавляется нужное число литер «*», в противном случае выводится только начальная часть вектора (или набора) заданной длины.

При вводе возможны следующие случаи:

— абсолютная величина вводимого числа превышает максимально возможную для той переменной, в которую оно вводится. В этом случае переменной простого формата присваивается пустой объект, а переменной строчного и нестандартного формата — нулевое значение. Аналогичные действия производятся при попытке ввода адресного значения по ⟨формату тегированного⟩;

— длина последовательности литер, вводимой по ⟨трафарету литерного⟩, не соответствует формату переменной или длине вектора, в которых происходит ввод. В этом случае введенная последовательность обрезается справа до нужной длины или заполняется литерами «*» до нужной длины.

Во всех случаях в журнал задачи *) выдается сообщение об ошибке.

Ошибка литеры. При возникновении ошибок этого класса в журнал задачи выдается соответствующее сообщение.

14. Групповые операции над векторами

В этом разделе описываются групповые операции над смежными векторами: различные варианты пересылок, поиск элемента, удовлетворяющего заданному условию, операции, связанные с переводом чисел из литерного представления в шестнадцатеричное и обратно (упаковка и распаковка).

⟨операции над массивами⟩ ::= ⟨пересылка⟩
| ⟨поиск⟩ | ⟨упаковка⟩

Здесь же рассматривается ⟨сравнение строк⟩.

Каждая из этих операций имеет в общем случае несколько операндов. В операциях пересылки основными являются ука-

*) Журнал задачи — это листинг сообщений системы о ходе решения задачи.

затель вектора назначения и указатель вектора источника; в операциях поиска и упаковки основным операндом является указатель вектора источника. Кроме того, в операциях можно задавать максимальное количество обрабатываемых элементов, эталон для сравнения и другие параметры.

Операции осуществляют циклическую обработку элементов векторов. На каждом шаге выполняются следующие действия: проверка условий окончания операции, обработка текущих элементов, продвижение указателей на один элемент. В связи с тем, что существует несколько условий окончания операции — исчерпание вектора источника, исчерпание вектора назначения, исчерпание заданного числа элементов, выполнение (или невыполнение) заданного отношения,— введены так называемые признаки, принимающие логические значения. По конечному состоянию признаков можно выяснить конкретные обстоятельства, имевшие место в момент окончания операции. В общем виде базовый алгоритм групповых операций выглядит следующим образом:

```
начало
    ф64 назначение := назначение, источник := источник ,
    макскол := макскол ;
    ... установить признаки в ложь ... ;
    до исчерпвект  исчерпмакс, отношение
    цикл
        если к = 0 то исчерпмакс !
        иначе если длина x = 0 или длина y = 0 то
            исчерпвект !
        иначе
            ... обработка y [0] , присваивание x [0] ...
            x := x [1] ; y := y [1 :] ; к := к — 1
        все
    повторить
    при
    исчерпвект :
        если длина x = 0 то
            ... установить признак переполнения тги в истину...
        все ;
        если длина y = 0 то
            ... установить признак источника тги в истину ...
        все ,
    отношение : ... установить признак отношения тго в истину...
    всесит ;
    % модифицировать параметры:
    ... назначение := x ; источник := y ; макскол := к ...
    ... выдать результат ...
конец
```

В отношении базового алгоритма следует сделать ряд уточнений.

1. Способ обработки источника, выдаваемый результат и обстоятельства, при которых возникает ситуация *отношение*, зависят от конкретной операции (см. пп. 14.1, 14.2).

2. Максимальное количество обрабатываемых элементов можно не задавать. В этом случае все действия, связанные с количеством, пропускаются. Аналогичным образом трактуется случай отрицательного количества.

3. В операциях поиска и упаковки вектор назначения не участвует. В этих случаях все действия, связанные с ним, пропускаются.

4. Заключительная модификация параметров производится только в том случае, если это явно указано в операции. Модификация задается путем добавления к параметру служебного слова мод.

Формат элементов вектора назначения и формат элементов вектора источника должны совпадать. Источник, если особо не оговорено, может быть представлен либо указателем с-вектора, либо объектом любого другого простого типа. Во втором случае ВРП объекта интерпретируется как вектор, элементы которого пронумерованы слева направо и имеют такой же формат, как элементы вектора назначения. В случае простого формата считается, что этот вектор состоит из одного элемента. В случае строчного формата считается, что количество элементов равно 64(ф1), 16(ф4), 8(ф8); в неполном наборе рассматривается только значащая часть.

14.1. Пересылка. Возможны следующие способы пересылки: простая пересылка (условная или безусловная), производящая копирование элементов вектора источника в вектор назначения, циклическое заполнение элементов вектора назначения одинаковым содержимым, пересылка литер с перекодировкой, пересылка шестнадцатеричных цифр с распаковкой в литеры. Способ пересылки задается формой ⟨источника пересылки⟩.

⟨пересылка⟩ ::=
⟨вектор назначения⟩⟨операция пересылки⟩
⟨источник пересылки⟩
{&& ⟨источник пересылки⟩} ...
⟨источник пересылки⟩ ::=
⟨безусловная пересылка⟩ | ⟨условная пересылка⟩
| ⟨перевод⟩ | ⟨распаковка⟩
| ⟨заполнение⟩
⟨вектор назначения⟩ ::= ⟨формула⟩ | мод ⟨переменная⟩
⟨операция пересылки⟩ ::= <:=

С помощью сцепления ⟨источников пересылки⟩ можно задать последовательность пересылок из различных источников в один вектор назначения. На каждом последующем этапе заполняется группа элементов вектора назначения, непосредственно следующая за группой, заполненной на предыдущем этапе. Сцепление обозначается парой символов &&.

Один или несколько параметров пересылки (вектор назначения, вектор источника, максимальное количество) можно передать в операцию «именем». Для этого перед соответствую-

щим параметром добавляется служебное слово **мод**. В этом случае соответствующей переменной присваивается модифицированное значение параметра (продвинутый указатель назначения или источника, количество элементов, оставшихся необработанными). Параметры ⟨источника пересылки⟩ модифицируются по окончании пересылки из этого источника. Переменная, задающая вектор назначения, модифицируется в конце операции.

Если ⟨пересылка⟩ играет роль ⟨формулы⟩, то в качестве результата она выдает продвинутый указатель вектора назначения.

Простая пересылка. Простая безусловная пересылка заканчивается при исчерпании вектора назначения и/или вектора источника и/или максимального количества. Условная пересылка может закончиться либо по тем же причинам, либо из-за несравнения текущего элемента вектора источника с эталоном.

⟨безусловная пересылка⟩ ::=
{⟨простой формат⟩}⟨вектор источника⟩
длиной ⟨максколичество⟩

⟨условная пересылка⟩ ::=
⟨безусловная пересылка⟩ **пока** ⟨индикатор отношения⟩
⟨первичное⟩

⟨вектор источника⟩ ::= ⟨первичное⟩ | **мод** ⟨переменная⟩

⟨максколичество⟩ ::= ⟨первичное⟩ | **мод** ⟨переменная⟩

⟨индикатор отношения⟩ ::= ⟨операция сравнения⟩ | **среди** | **не среди**

Безусловная пересылка определена для простых и строчных форматов. В случае простого формата его необходимо указывать явно. Способ обработки состоит в копировании элемента вектора источника в вектор назначения: $x[0] := y[0]$.

Условная пересылка определена только для строчных форматов (⟨простой формат⟩ в этом случае нельзя указывать). Способ обработки определяется следующим условным предложением:

если $y[0]$ *R эталон* **то** $x[0] := y[0]$ **иначе** *отношение* **!** **все**,

где R обозначает ⟨индикатор отношения⟩, а *эталон* — это значение ⟨первичного⟩.

Заполнение. ⟨Заполнение⟩ присваивает элементам вектора назначения одно и то же значение: $x[0] := y$, где y является значением ⟨первичного⟩.

⟨заполнение⟩ ::=
{⟨простой формат⟩} **заполи** ⟨первичное⟩
{**длиной** ⟨максколичество⟩}

Операция заполнения определена для простых и строчных форматов. В случае простого формата его необходимо указывать явно.

Перевод. ⟨Перевод⟩ пересылает элементы вектора источника с промежуточной перекодировкой: $x[0] := t[y[0]]$, где t — таб-

лица перекодировки (задается значением ⟨первичного⟩).

⟨перевод⟩ ::=
перевод ⟨вектор источник⟩ {**длиной** ⟨максколичество⟩}
по ⟨первичное⟩

Пересылка с переводом определена для литерного формата. Таблица перекодировки также должна быть литерным *с*-вектором.

Распаковка. ⟨Распаковка⟩ преобразует шестнадцатеричные цифры источника в соответствующие литеры и пересылает их в вектор назначения.

⟨распаковка⟩ ::=
⟨операция распаковки⟩ ⟨первичное⟩
{**длиной** ⟨максколичество⟩}
⟨операция распаковки⟩ ::= **распак** | **распакзн**

ВРП значения ⟨первичного⟩ независимо от типа рассматривается как последовательность шестнадцатеричных цифр. В неполном наборе рассматривается только значащая часть. Вектор назначения может быть не только литерным, но и цифровым. В последнем случае цифры источника пересылаются без преобразования.

Преобразование цифр в литеры состоит в том, что к каждой цифре *z* слева добавляется шестнадцатеричный код *f*. В результате образуется соответствующая цифре литера 4"*fz*".

Операция **распак** выполняет пересылку без учета знака, а операция **распакзн** дополнительно вставляет знак, исходя из текущего значения признака отношения. При распаковке в литеры знак кодируется в поле зоны последней распакованной цифры: 4"*cz*" — положительный знак (**тго**=**истина**), 4"*dz*" — отрицательный знак (**тго**=**ложь**). При пересылке в цифровой вектор в качестве первой цифры пересылается знак: 4"*c*" — плюс, 4"*d*" — минус. В этом случае длина результирующей цифровой строки увеличивается на 1. В параметре, задающем максимальное количество пересылаемых цифр, дополнительный элемент не учитывается.

14.2. Операции поиска. Операция ⟨поиска по строке⟩ выполняет поиск элемента строчного формата, удовлетворяющего заданному отношению. Кроме того, существует операция поиска с маскированием.

⟨поиск⟩ ::= ⟨поиск по строке⟩ | ⟨поиск по маске⟩

Поиск по строке. Эта конструкция выполняет либо поиск элемента, удовлетворяющего заданному отношению, либо поиск элемента с заданным номером, либо поиск элемента, удовлетворяющего одному из этих условий. В случае проверки отношения способ обработки задается следующим условным предложением:

если *y* [0] *R эталон* **то** *отношение* | **все**

Здесь использованы те же обозначения, что и в п. 14.1. Поиск может прекратиться при обнаружении искомого элемента или при исчерпании вектора источника.

⟨поиск по строке⟩ ::= ⟨указатель источника⟩
от ⟨условие поиска⟩
⟨условие поиска⟩ ::=
⟨индикатор отношения⟩⟨первичное⟩
| ⟨максколичество⟩{**либо** ⟨индикатор отношения⟩
⟨первичное⟩}
⟨указатель источника⟩ ::= ⟨формула⟩ | **мод** ⟨переменная⟩

Эталон для сравнения задается значением ⟨первичного⟩, а номер искомого элемента компонентой ⟨максколичество⟩. Операция определена для векторов, состоящих из элементов строчного формата. Значением ⟨указателя источника⟩ должен быть указатель с-вектора, остальные типы в данном случае запрещены. Если ⟨поиск по строке⟩ играет роль ⟨формулы⟩, то результатом операции является продвинутый указатель вектора источника. В случае, если операция закончилась не из-за исчерпания источника, этот указатель описывает часть вектора источника, начинающуюся с искомого элемента.

Если в конструкции предписана модификация параметра, задающего номер искомого элемента, необходимо учитывать, что в соответствии с базовым алгоритмом модифицированное значение определяет разницу между первоначальным значением и числом пропущенных элементов, а не номер найденного элемента. Сравнение производится при совпадении типов.

Поиск с маскированием. Поиск с маскированием определен для векторов, состоящих из элементов формата ф64. В процессе поиска очередной элемент вектора маскируется, т. е. логически умножается на заданную маску, и результат сравнивается с маскированным эталоном. В маскировании участвует 70 разрядов значения: 6 разрядов типа и 64 информационных разряда.

Если заданное отношение не выполняется, производится сравнение следующего элемента. В качестве следующего элемента выбирается соседний элемент вектора или следующий элемент списка в зависимости от того, что предписано операцией. Операция может закончиться после того, как найден нужный элемент, или после исчерпания вектора (алгоритм этой операции отличается от базового алгоритма других групповых операций).

⟨поиск по маске⟩ ::=
поиск (⟨выражение⟩,⟨выражение⟩{,{⟨выражение⟩,}
⟨выражение⟩})
⟨способ выборки⟩⟨операция сравнения⟩⟨первичное⟩
⟨способ выборки⟩ ::= **до** | **вниздо** | **списокмдо** | **цепьюдо**

Значения первых двух ⟨выражений⟩ создают соответственно вектор и индекс элемента, с которого начинается поиск. Следующие два параметра предназначены для формирования маски. Маска типа задается в правых 6 разрядах значения третьего ⟨выражения⟩, а маска информационной части задается в 64 разрядах значения четвертого ⟨выражения⟩. Эталоном для сравнения является значение ⟨первичного⟩. Последние три зна-

чения могут иметь любой тип. По умолчанию маска типа полагается равной нулю, т. е. в процессе поиска тип элементов и тип эталона игнорируются. Маска информационной части по умолчанию заполнена единицами.

Возможны четыре способа выборки следующего элемента.

Служебное слово до(вниздо) задает приращение (уменьшение) индекса на каждом шаге на 1. Если поиск прошел успешно, операция выдает индекс найденного элемента. В противном случае признак **тги** устанавливается в истину, и выдается индекс, выходящий за границу массива. При прямом направлении обработки он равен длине вектора, а при обратном направлении он равен числу $2^{20} - 1$.

Служебные слова **списком до** и **цепью до** задают поиск по связанному списку. Индекс следующего элемента списка выбирается из поля [19:20] текущего элемента. В способе **цепью до** ищется либо элемент, для которого выполняется заданное отношение, либо элемент, содержащий стоп-бит.

В обоих случаях первый элемент в сравнении не участвует. Если поиск по списку прошел успешно, то выдается индекс элемента, указывающего на найденный элемент. Индекс имеет отрицательный знак в случае, если при способе выборки **цепью до** поиск прекратился на элементе, не удовлетворяющем отношению, но содержащем стоп-бит. В процессе поиска по списку возможны выход за границу массива и зацикливание. В первом случае возникает ситуация *границамассива*, во втором случае — ситуация *исчерпвремяопрц*.

14.3. Сравнение строк. Операция ⟨сравнения строк⟩ производит поэлементное сравнение векторов, состоящих из элементов строчного формата, и выдает логическое значение **истина** или **ложь**. Операция продолжается до исчерпания одного из векторов.

В случае литерных и цифровых строк результатом операции является результат сравнения первой встретившейся пары неравных элементов. При сравнении битовых строк значение **истина** выдается только в том случае, если заданное соотношение имеет место для всех сравниваемых пар. Это свойство имеет теоретико-множественное толкование: если строки x и y задают подмножества некоторого множества (элементы строки, соответствующие элементам подмножества, имеют значение **истина**, а остальные — **ложь**), то $x=/y$ проверяет эквивалентность подмножеств, $x<=/y$ повторяет включение x в y, а $x>=/y$ проверяет включение y в x.

⟨сравнение строк⟩ ::=

⟨вектор источник⟩⟨операция сравнения⟩/

⟨вектор источник⟩

[**длиной** ⟨максколичество⟩]

⟨операция сравнения⟩ ::= < | > | = | <> | <= | >=

Правый операнд может принимать такие же значения, как источник в операциях пересылки. Значением левого операнда должен быть указатель с-вектора. Установка признаков и заключительная модификация параметров имеют такую же трактовку, как и в других групповых операциях. По завершении

операции состояние признака отношения совпадает с результатом операции.

В случае, если длина какого-либо из исходных векторов и/или начальное значение ⟨максколичества⟩ оказались равными нулю, векторы считаются равными.

14.4. Упаковка. Операция упаковки предназначена в основном для преобразования целого числа из литерного представления в набор шестнадцатеричных цифр.

⟨упаковка⟩ ::= **пак** ⟨вектор источник⟩
{длиной ⟨максколичество⟩**}**

Операция определена для литерных и цифровых векторов. Если тип операнда отличен от указателя с-вектора, то операнд рассматривается как литерный вектор; в неполном наборе рассматривается только значащая часть.

Преобразование литер состоит в том, что в каждом элементе отбрасывается «зона» — левые четыре разряда. Шестнадцатеричные цифры переносятся в результирующий набор без преобразования. Результатом операции является цифровой набор, длина которого равна количеству обработанных элементов. Если это количество превышает число 16, то возникает ситуация *неверныйоперанд*. Цифры расположены в значащей части набора слева направо в порядке обработки.

Обрабатываемая строка может содержать знак числа, закодированный в такой форме, как это описано в операции ⟨распаковки⟩ **распакзн**. В этом случае операция упаковки производит обратное преобразование и восстанавливает состояние признака отношения. Знаковый элемент исходной цифровой строки не учитывается в параметре ⟨максколичество⟩.

14.5. Признаки. Значение ⟨признаков⟩ устанавливается и анализируется в процессе выполнения операций перевода чисел и групповых операций над векторами. По текущему значению признаков можно судить о том, какие обстоятельства привели к окончанию групповой операции. ⟨Признак⟩ может иметь одно из логических значений: **истина** или **ложь**.

⟨признак⟩ ::= **тго** | **тгп** | **тги**

Служебные слова имеют следующий смысл: **тго** — признак отношения, **тгп** — признак переполнения вектора назначения, **тги** — признак исчерпания источника.

Явным образом можно установить только признак отношения. Это делается с помощью стандартной функции *тгз*; *тгз*(x) устанавливает признак отношения в 1 (**истина**), если значение x — отрицательное число, и в 0 (**ложь**) — в противном случае.

Перед инициированием выполнения базированного блока текущие значения признаков запоминаются, а по окончании выполнения восстанавливаются.

15. Преобразование формы массива

В этом разделе описываются средства языка, позволяющие из элементов одного массива составить другой массив, форма которого (число измерений, длина измерений и пр.) отличает-

ся от исходной. Следует отметить, что при этом дополнительная память не выделяется и элементы первого массива не копируются, а лишь формируется паспорт, по-новому описывающий нужную часть исходного массива. Таким образом, составление нового массива заключается в наложении нового паспорта на исходный массив. В связи с этим новый массив в этом разделе называется наложенным массивом. Исходный паспорт в общем случае сохраняется. По этой причине одновременно могут существовать несколько паспортов, различным образом описывающих один и тот же массив.

15.1. Формирование паспорта. Конструкция ⟨формирование паспорта⟩ вводит наложенный массив, который образуется из исходного путем следующих основных преобразований: сдвиг начала измерений, сокращение длин измерений, увеличение шага по измерениям (растяжение координатных осей), сокращение числа измерений, перестановка измерений местами.

⟨формирование паспорта⟩ ::=

формавм ({⟨описатель⟩}[⟨список идентификаторов⟩] = ⟨описатель⟩[⟨список диапазонов или индексов⟩])

⟨описатель⟩ ::= ⟨идентификатор⟩ | ⟨закрытое выражение⟩

⟨диапазон или индекс⟩ ::= ⟨суперпозиция⟩{=⟨диапазон⟩}

⟨суперпозиция⟩ ::= ⟨слагаемое⟩{⟨знак⟩⟨слагаемое⟩} ...

⟨слагаемое⟩ ::= ⟨сомножитель⟩{*⟨сомножитель⟩}

Исходный массив задается значением ⟨описателя⟩, находящегося в правой части. Значение ⟨описателя⟩ в левой части задает объект типа паспорт. В этот объект копируется вновь сформированный паспорт. В целом смысл конструкции состоит в том, что она задает соответствие между элементами наложенного и исходного массивов. При этом используется метод, схожий с методом формальных параметров. ⟨Список идентификаторов⟩ в левой части действует аналогично объявлению формальных параметров. В нем вводятся идентификаторы, с помощью которых в правой части обозначаются индексы наложенного массива.

В каждой индексной позиции правой части может находиться линейная ⟨суперпозиция⟩ формальных индексов. В этом случае ⟨идентификаторы⟩ формальных индексов являются ⟨сомножителями⟩. Например:

формавм (*a* [*i*] = *b* [2 * *i*])

где *i* — формальный индекс, означает, что элементы наложенного массива *a* — это четные элементы исходного массива *b*.

Чтобы определить элемент исходного массива, соответствующий конкретному элементу наложенного массива, надо индексы последнего подставить как фактические параметры в правую часть. Получившиеся при этом индексы по исходному массиву не должны выходить за разрешенный диапазон. Нижняя граница задается значением первого ⟨выражения⟩ в ⟨диапазоне⟩, а длина диапазона — значением второго ⟨выражения⟩. Если первое (второе) ⟨выражение⟩ опущено, то подразумевается, что нижняя (верхняя) граница совпадает с нижней (верхней)

границей исходного массива по данному измерению. ⟨Диапазон⟩ может быть полностью опущен вместе со знаком равенства. Это означает, что разрешенный диапазон совпадает с диапазоном изменения индекса исходного массива по данному измерению.

Если в индексной позиции находится ⟨суперпозиция⟩, не зависящая от формальных индексов, то тем самым индекс по данному измерению исходного массива фиксируется и полагается равным значению ⟨суперпозиции⟩. При этом число измерений, очевидно, уменьшается на 1. В данном случае ⟨диапазон⟩ не задается. Например,

формавм (*a* [*i*] = *bb* [2 * *i* , 0])

означает, что элементы наложенного вектора *a* — это четные элементы нулевого столбца исходной матрицы *bb*.

Число измерений наложенного массива не должно превышать числа измерений исходного массива. Каждый формальный индекс должен входить по крайней мере в одну ⟨суперпозицию⟩, но может встречаться и в нескольких.

⟨Идентификатор⟩ формального индекса может находиться в правой части ⟨формирования паспорта⟩ только на месте ⟨сомножителя⟩ ⟨суперпозиции⟩. Причем, если один ⟨сомножитель⟩ какого-либо ⟨слагаемого⟩ — формальный индекс, то другой ⟨сомножитель⟩ не может быть таковым, т. е. ⟨суперпозиция⟩ линейно зависит от формальных индексов.

Компоненты ⟨формирования паспорта⟩, отличные от ⟨идентификаторов⟩ формальных индексов, выполняются в момент выполнения всей конструкции. Значением ⟨описателя⟩ в левой части является указатель, описывающий объект типа паспорт *n*-мерного массива, где *n* — число формальных индексов. Если левый ⟨описатель⟩ опущен, то такой объект создается неявно и локализуется в ближайшем блоке.

Значением конструкции является указатель паспорта наложенного массива. При заданном левом ⟨описателе⟩ конструкцию можно использовать в роли ⟨оператора⟩.

Случай параллелепипеда. Пусть в левой части вводится *n* формальных индексов, а каждый ⟨диапазон или индекс⟩ в правой части имеет вид $m*i+c=n:\kappa$, где *i* — один из формальных индексов (*n*, *m*, *c* и *к* могут быть опущены; если члену суперпозиции предшествует отрицательный знак, то он учтен в *m* или *c*). Тогда элементы наложенного массива составляют в *n*-мерном координатном пространстве прямоугольный параллелепипед *).

Пример.

```
конст
    am = формавм ([i , j] = a[j , i]) , % am — паспорт транс-
                                        % понированной мат-
                                        % рицы a
    адиаг = формавм ([i] = a[i , i]) , % адиаг — паспорт диа-
                                        % гонали матрицы a
```

*) Обычный в-массив, создаваемый с помощью генератора в-массива, всегда является параллелепипедом.

```
аобрдиаз = формавм ([i] = a [i , (l — 1) — i])
                                        % аобрдиаз —
                                        % паспорт
                                        % обратной
                                        % диагонали
                                        % матрицы a
                                        % размером
                                        % l * l
минор = формавм ([i , j] = a[i = 0 : к , j = 0 : к])
                        % паспорт минора к-го порядка.
```

Индекс i может пробегать следующий диапазон значений: $(н - с)/:т \leqslant i \leqslant (н + к - 1 - с)/:т$

Если i встречается в нескольких суперпозициях, таких неравенств несколько, и диапазон изменения индекса определяется путем выбора минимальной верхней границы и максимальной нижней. Результирующая нижняя граница должна быть не больше верхней.

Как видно из предыдущего, нижняя граница индексов наложенного массива может оказаться отличной от нуля и, в частности,— отрицательной. Язык допускает индексацию таких в-массивов, но средства, позволяющие по паспорту установить нижние границы, отсутствуют. Однако, используя атрибут *нигрнуль*, можно установить все нижние границы в нуль и обрабатывать его обычным образом, предполагая, что каждый индекс изменяется, начиная от нуля в сторону положительных значений.

Паспорт прямоугольного параллелепипеда обладает следующими свойствами:

1. Для него определены значения всех атрибутов паспорта. В частности, путем запроса атрибута *длинизм* можно определить длины по измерениям.

2. К нему применимы все конструкции и операции, связанные с в-массивом.

Случай произвольной суперпозиции. Это случай, когда по крайней мере в одну из суперпозиций входит более одного формального индекса. Такое наложение используется лишь с целью оптимизации. Оно позволяет любую индексацию массива внутри цикла представить в нормальной форме. Нормальной формой называется индексация вида $a[i1, i2, \ldots, in]$, где a — идентификатор константы или переменной, а $i1, i2, \ldots, in$ параметры циклов. В этом случае транслятор обеспечивает генерацию наиболее эффективного кода.

Пример.

Во вложенном цикле происходит обращение к элементам массива a:

```
для i до ni
цикл
    для j до nj
    цикл
```

```
    ... a [2 * i + j , j + 1] ...
  повторить
повторить
```

Используя ⟨формирование паспорта⟩, можно перейти к индексации в нормальной форме:

```
конст aonm = формавм ([i1 , j1] = a [2 * i1 + j1 , j1 + 1]) ;
для i до ni
цикл
  для j до nj
  цикл
    ... aonm [i , j] ...
  повторить
повторить
```

В случае произвольной суперпозиции особенность результирующего паспорта состоит в следующем:

— для него не определен ряд атрибутов,

— его нельзя использовать в правой части конструкции ⟨формирование паспорта⟩.

— для него не определены стандартные функции над паспортами (см. ниже).

15.2. Стандартная функция *пввект*.

пввект (*a*, *b*, *bu*). Функция формирует паспорт выстроенного вектора, наложенного на элементы n-мерного массива $b[i_1, i_2, \ldots, i_{n-1}, i_n]$, где индексы $i_1, i_2, \ldots, i_{n-1}$ фиксированы, а i_n пробегает все допустимые значения. Индексы $i_1, i_2, \ldots, i_n$ являются значениями элементов вектора *bu*, содержащего ровно $n-1$ элемент:

$bu[0] = i_1$, $bu[1] = i_2, \ldots, bu[n-2] = i_{n-1}$.

Параметр *a* задает объект типа паспорт (одномерного массива), куда и копируется результирующий паспорт: он же является результатом функции.

Эта функция в сочетании с атрибутами паспорта *чслизм* (число измерений) и *длинизм* (длины измерений) позволяет обрабатывать выстроенный массив, число измерений которого статически неизвестно.

Функция может вызываться с одним параметром: *пввект*(*b*) В этом случае *b* — паспорт одномерного в-массива. Он же является значением функции. Этот вариант предназначается для того, чтобы квалифицировать значение выражения *b* (в частности,— формальный параметр, см. п. 6.2) как паспорт в-вектора.

15.3. Генератор паспорта. ⟨Генератор паспорта⟩ позволяет создать объект типа паспорт. Этот паспорт является подвижным, т. е. его содержимое можно менять таким образом, чтобы он в разное время описывал разные массивы.

```
⟨генератор паспорта⟩ ::=
    ⟨локализация⟩ [{ , } ...]
  | ⟨локализация⟩[⟨список выражений⟩]
```

Пример.

```
начало
    конст a = лок [ , ] ; % создать паспорт
        ...
        % наложить паспорт a на транспонированную
        % матрицу b
        формавм (a [i , j] = b [j , i]) ;
        ...
        % наложить паспорт a на транспонированную
        % матрицу c:
        формавм (a [i , j] = c[j , i]) ;
        ...
конец
```

Значение атрибута *чслизм* созданного паспорта (число измерений, см. атрибуты паспорта) равно количеству индексных позиций между скобками. Если ⟨список выражений⟩ опущен, то длины по измерениям (атрибут *длинизм*) первоначально равны 0. В противном случае (а) они равны значению соответствующих ⟨выражений⟩ и (б) содержимое созданного паспорта можно менять только путем модификации атрибута *базвект*.

Постоянные и подвижные паспорта. Паспорта, создаваемые ⟨генератором в-массива⟩ и неявно — ⟨формированием паспорта⟩ (при опущенном ⟨описателе⟩ в левой части конструкции), являются постоянными, т. е. они описывают один и тот же массив. Их нельзя накладывать на другой массив с помощью ⟨формирования паспорта⟩ или путем модификации атрибута *базвект*.

В отличие от этого содержимое паспорта, созданного с помощью ⟨генератора паспорта⟩, изменять можно (с учетом указанного выше ограничения).

16. Участок базированной области

⟨Описание базы⟩ вводит идентификатор, который обозначает указатель, описывающий участок текущей базированной области памяти (БОП). Этот участок представляет собой подобласть БОП, в которой расположена переменная часть параметров, или подобласть БОП, распределяемую в результате выполнения ⟨описаний⟩, непосредственно следующих за ⟨описанием базы⟩. Первый случай связан с использованием ⟨описания базы⟩ в списке формальных параметров процедуры (см. п. 11.1); во втором случае ⟨описание базы⟩ используется среди ⟨описаний⟩ в ⟨закрытом предложении⟩. Выполнение самого этого описания не приводит к распределению дополнительной памяти.

⟨описание базы⟩ ::= база ⟨идентификатор⟩

Необходимо учитывать, что результирующий указатель описывает участок БОП как смежный вектор элементов формата ф32; при необходимости формат можно преобразовать с помощью конструкции ⟨подмассив⟩ (см. примеры в п. 11 гл. 1).

17. Текстовые макросы

⟨Описание текстов⟩ и ⟨подстановка текста⟩ представляют собой простейшую форму макросредств языка. Описание связывает идентификатор с текстом ⟨предложения⟩, находящегося в правой части. ⟨Подстановка текста⟩ выполняется в процессе трансляции путем копирования текста ⟨предложения⟩ в место подстановки.

⟨описание текстов⟩ ::= текст ⟨список идентификации текстов⟩
⟨идентификация текста⟩ ::=
⟨идентификатор⟩{({⟨список идентификаторов⟩})}=
⟨предложение⟩

⟨Предложение⟩ может быть ⟨выражением⟩, ⟨оператором⟩ либо ⟨переменной⟩. Выбор зависит от того, какие синтаксические формы разрешены в тех местах программы, куда подставляется текст данного ⟨предложения⟩.

Текст может быть параметризован с помощью ⟨списка идентификаторов⟩ формальных параметров текста. Областью действия формальных параметров является ⟨предложение⟩ в правой части. Каждое вхождение формального параметра внутри ⟨предложения⟩ рассматривается как ⟨подстановка текста⟩ без параметров.

⟨подстановка текста⟩ ::=
⟨идентификатор⟩ {({⟨список предложений⟩})}
| ⟨идентификатор⟩ [⟨список предложений⟩]

⟨Подстановка текста⟩ выполняется следующим образом. ⟨Предложение⟩, связанное с ⟨идентификатором⟩ текста, заключается в скобки и копируется в место подстановки. В процессе копирования рекурсивно выполняются все внутренние подстановки текстов. В частности, аналогичным образом выполняется подстановка фактических параметров. При подстановке текста, как и при вызове процедуры, идентификаторы сохраняют свой смысл: идентификаторы входящие в ⟨предложение⟩, имеют тот смысл, который им был приписан в месте описания текста, и идентификаторы, использованные в фактических параметрах, имеют тот смысл, который им был приписан в месте подстановки текста.

Число фактических параметров текста должно совпадать с числом формальных параметров. Пустые скобки в ⟨подстановке текста⟩ используются в том случае, если они были заданы в левой части описания этого текста. Это позволяет одинаковым образом обозначать ⟨вызов⟩ процедуры и ⟨подстановку текста⟩.

Выполнение ⟨подстановки текста⟩ не должно приводить к рекурсивной подстановке того же самого текста.

18. Атрибуты объектов

Конструкция ⟨запрос атрибута⟩ выдает текущее значение заданного атрибута, а конструкция ⟨модификация атрибутов⟩ изменяет значение указанных атрибутов.

Атрибутам поставлены в соответствие номера атрибутов — стандартные целочисленные константы, с помощью которых атрибуты обозначаются при запросе и модификации. Идентификаторы этих констант перечислены в гл. 4. Там же описана семантика атрибутов.

⟨запрос атрибута⟩ ::=

читатр (⟨выражение⟩ , {⟨уточнение⟩ , } ⟨выражение⟩)

⟨уточнение⟩ ::= ⟨выражение⟩

Значение первого ⟨выражения⟩ — это указатель, приводящий к объекту. Значение последнего ⟨выражения⟩ задает номер атрибута.

⟨модификация атрибутов⟩ ::=

запатр (⟨выражение⟩ , {⟨уточнение⟩ , }
⟨список установок отрибутов⟩)

⟨установка атрибута⟩ ::= ⟨выражение⟩ : ⟨выражение⟩

В этой конструкции первое ⟨выражение⟩ трактуется, как и в предыдущей. В ⟨списке установки атрибутов⟩ перечисляются изменяемые атрибуты: значение первого ⟨выражения⟩ элемента списка задает номер атрибута, а значение второго ⟨выражения⟩ — новое значение атрибута.

⟨Модификация атрибутов⟩ может использоваться и в роли ⟨оператора⟩ и как ⟨выражение⟩. Во втором случае ее значением является значение первого параметра.

В обеих конструкциях может присутствовать компонента ⟨уточнение⟩. Она используется в ряде случаев при запросе атрибутов в-массива (см. § 9 гл. 4), а также при запросе/модификации атрибутов внешних объектов и оперативных объектов связи (см. § 3 гл. 3).

Примеры.

```
читатр (а, чслизм)          % число измерений массива
читатр (а, 1, длинизм)      % длина первого измерения
                            % массива а
запатр (ф, типфайла : 3)    % изменение атрибута
                            % типфайла у файла ф
```

Ряд дополнительных сведений о запросе и модификации атрибутов внешних объектов и оперативных объектов связи рассмотрен в § 3 гл. 3.

19. Описание и использование меток

В этом разделе описываются средства языка, предназначенные для программирования с использованием оператора перехода.

⟨Описание метки⟩ является предварительным описанием идентификатора, используемого в качестве ⟨метки⟩ некоторого предложения.

⟨описание меток⟩ : : = метка ⟨список идентификаторов⟩

⟨метка⟩ ::= ⟨идентификатор⟩^:

⟨Описание метки⟩ должно предшествовать ⟨метке⟩ и любому использованию ⟨идентификатора⟩ метки. ⟨Описание метки⟩ и ⟨метка⟩ должны находится на одном лексикографическом уровне, т. е. если ⟨описание метки⟩ дано в начале закрытого или последовательного предложения, то соответствующая ⟨метка⟩ не может встречаться внутри вложенного закрытого или последовательного предложения с описаниями. ⟨Идентификатор⟩ метки трактуется как идентификатор константы, значением которой является соответствующая метка перехода.

Оператор перехода передает управление на то предложение, которое помечено меткой, являющейся значением его операнда.

⟨переход⟩ ::= **на** ⟨одноместная формула⟩

При выполнении глобального перехода прекращается выполнение всех промежуточных блоков; попутно уничтожаются локализованные в них объекты. *

20. Примеры

В этом параграфе приводятся примеры программ. В примерах 1—12 исходные данные задаются в программе, а результаты выводятся на печать. В примере 13 представлено задание, в котором вызывается программа с двумя входными файлами-параметрами.

Пример 1. Работа с циклом. Вычисление суммы ряда.

$h(n) = 1 + 1/2 + 1/3 + \ldots + 1/n$

```
начало
    конст печ = позп (ацпу);
    процедура h = функция (n)
        начало
            ф64 сум := 0 ;
            для i от n вниздо 1
            цикл
                сум := сум + 1/i
            повторить ;
            сум
        конец;
    запф(печ , h (10) : ''сумма =''  g (0 , 13))
конец
```

Пример 2. Поиск *min* и *max* в массивах *a* и *b*.

```
начало
    конст
        печ = позп (ацпу) ,
        a = мконст ф64 (8 , 3 , 4 , 6 , 1) ,
        b = мконст ф64 (0 , 5 , 7 , 13 , 16 , 12 , 5 , 8 , 2 , 11) ;
    ф64 minn , maxx ;
```

```
процедура minmax = проц (t , min , max)
    начало
        min := max := t[0] ;
        для i до длина t — 1
        цикл
            если конст v = t [i] ; v > max то max := v
            инес v < min то min := v
            все
        повторить
    конец ; % конец процедуры minmax
  minmax (a , имя minn , имя maxx) ;
  вапф (печ , minn : ''min = ''2 zd2x , maxx : ''max = ''2zd) ;
  minmax (b , имя minn , имя maxx) ;
  вапф (печ , minn :'' min = '' 2zd2x , maxx :'' max = '' 2zd)
конец
```

Пример 3. Выход из цикла по ситуации. Вычисление косинуса

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \ldots + (-1)^{\kappa}\frac{x^{2\kappa}}{(2\kappa)!}$$

до $a_\kappa/\cos(x) < eps$, где a_κ — κ-й член суммы.

```
начало
    конст печ = позп(ацпу) ;
    процедура cos = функция (x)
        начало
            конст eps = 1e — 14 ;
            ф128  y := 1 , ак := 1 ; ф64 к := 0 ;
        до отношмнш
        цикл
            конст хкb = x * x ;
            текст abs (x) = x * знак x ;
            если abs (ак) < eps * abs (y) то отношмнш ! все ;
            к := к + 2 ;
            ак := —ак умнд хкb / (к * (к — 1)) ;
            y := y + ак
        повторить ;
            вапф (печ , к / : 2 : ''к = '' 2zd) ; % число итераций
            ф64окр y
        конец ;
  вапф (печ , 1.57 : ''cos (''g (0 , 4)'') = '' , cos (1.57) : g(0, 4))
конец
```

Пример 4. Выход из цикла по ситуации и со значением. В примере вычисляется $z = v ** e$, где e — целое.

```
начало
   конст печ = позп (ацпу) ;
   процедура возвстеп = функция (v , е)
      до исчерпстеп
      цикл ф64 z := 1 ;
         если е = 0 то исчерпстеп ! (z)
         иначе
            до нечет
            цикл
               если е . [0] то нечет !
               иначе v := v * v
               все ;
             . е := е /: 2
            повторить ;
            z := z * v
         все ;
        е: = е — 1
      повторить
      при исчерпстеп (zz) : zz % результат
      всесит ;                  % конец возвстеп
   запф (печ , возвстеп (5 ; 3) : ''5**3='' g (0))
конец
```

Пример 5. Работа с битовыми векторами. Решето Эратосфена.

```
начало
   процедура решето = проц (п)
      начало
         конст печ = позп(ацпу) , реш = лок вект [п] ф1;
         ф64 к := 1 ;
         запф (печ , 1 : 7zd) ;
         реш <:= заполн истина ;
         для i от 2 до п — 1
         цикл
            если реш [i] то
               запф (печ , i : 7zd) ;
               если (к := к + 1) > 16 то % перевод строки
              к := 1 ; запф (печ , : l)
               вес ;
               до верхгран
               цикл ф64 j := i ;
               реш [j] := ложь ;
               если (j := j + i) > (п — 1)
               то верхгран ! все
               повторить
```

```
            все
          повторить
        конец ; % процедуры решето
    решето (10000)
конец
```

Пример 6. Работа с цифровыми и литерными векторами.

```
начало
    процедура отрстеп2 = проц (n)
      начало % печать числа 2 ** (—n)
        конст
          m = лок вект [n] ф4 ,
          s = лок вект [n] ф8,
          печ = позп (ацпу);
        для к до n — 1
        цикл % от 2 ** (—1) до 2 **(—n)
          % к первых цифр частного
          для i до к — 1
          цикл ф64 r := 0 ;
            % алгоритм деления «столбиком»:
             r := 10 * r + m [i] ;% остаток от предыд.
                                  % итерации +
                                  % следующая цифра де-
                                  % лимого
            m [i] := r /: 2 ; % след. цифра частного
            r := r остат 2 ; % остаток
            s [i] := ''0'' + m [i] % цифра частного в литеру
        повторить ;
         % к + 1 - я цифра частного :
          m [к] := 5 ;
          s [к] := ''5'' ;
          запф (печ , к + 1 : ''1/2**'' 2zd'' = 0.'' ,
          s [: к + 1] , : l)
        повторить
    конец ; % процедуры отрстеп2
  отрстеп2 (10)
конец
```

Пример 7. Вычисление отрицательных степеней числа 2 с использованием пересылки цифр с распаковкой в литеры.

```
начало
  процедура отрстеп2 = проц (n)
    начало % печать числа 2 ** (—n)
      конст
        m = лок вект [n] ф4
```

```
        s = лок вект [n] ф8,
        печ = позп (ацпу) ;
      для к до n — 1
      цикл % от 2 ** (—1) до 2 ** (—n)
      для i до к — 1
        % к первых цифр частного:
        цикл ф64 r := 0 ;
          % алгоритм деления «столбиком»
          r := 10*r + m[i]  % остаток от предыдущей
                            % итерации + следующая
                            % цифра делимого
          m[i] := r /: 2 ;  % след. цифра частного,
           r := r остат 2   % остаток
        повторить ;
        % к + 1-я цифра частного :
        m [к] := 5 ;
        % преобразование в литерное
        % (не более 16 цифр ! , см. п. 14. 1) :
        s <:=распак (m [ : к + 1]) @ ;
        вапф (печ , к + 1 : ''1/2**'' 2zd'' = 0.'' ,
        s [: к + 1] , : l)
      повторить
    конец ; % отрстеп2
  отрстеп2 (10)
конец .
```

Пример 8. Сортировка массива методом простой вставки.

```
начало
  конст печ = позп (ацпу) , b = лок вект [5] ф64 ;
  процедура сортпр = проц (a)
    для i от 1 до длина a — 1
    цикл % слева от i-го элемента
         % ищется меньший или равный
         % a [i] ; затем a[i] вставляется перед ним
      ф64 ai ;
      ai := a [i] ; % сохранить a[i]
      до найден
        (для j от i — 1 вниздо 0
        цикл
          если a [j] <= ai то найден ! (j)
          иначе % сдвинуть j-й вправо
            a [j + 1] := a [j]
          все
        повторить ;
```

```
            найден ! (— 1))
        при
            найден (к) : а [к + 1] := ai
        всесит
    повторить ; % сортпр
  b <: = ф64 мконст ф64 (5 , 3 , 3 , 4 , 0) ; % заполнить b
  запф (печ , : ''исходный массив :'' , b : 10g (10)) ;
  сортпр (b) ;
  запф (печ , : ''результирующий массив:'' , b : 10g (10))
конец
```

Пример 9. Сортировка массива методом двоичной вставки.

```
начало
  конст b = лок вект [5] ф64, печ = позп (ацпу);
  процедура сортдв = проц (а)
    для i от 1 до длина а — 1
    цикл % место для i-го элемента
      ф64 ai ;      % ищется методом двоичного поиска
      ai := а [i] ; % сохранить а [i]
      до пересеч
      цикл
        ф64 m ,     % середина отрезка
        l := 0 ,    % левая граница
        r := i — 1 ; % правая граница
        если l > r то пересеч ! (l)
        иначе
          m := (l + r) /: 2 ;
          если ai < а [m] то r := m — 1
          иначе l := m + 1
          все
        все
      повторить
      при пересеч (ll) :
        для j от i — 1 вниздо ll
        цикл % сдвинуть элементы
          а [j + 1] := а [j]
        повторить ;
        а [ll] := ai
      всесит
    повторить ; % сортдв
  b <:= ф64 мконст ф64 (20 , 21 , 16 , 18 , 16) ;
  запф (печ , : ''исходный массив:'' , b : 10g (10)) ;
```

```
    сортдв (b);
    запф (печ , : ''результирующий массив:'' , b : 10g (10))
конец
```

Пример 10. Рекурсивные процедуры. Быстрая сортировка.

```
начало
  конст b =лок вект [5] ф64 , печ = позп (ацпу) ;
  процедура быстрсорт = проц (a)
    начало
      процедура раздел = проц (l , r)
        начало
          ф64 i := l , j := r ;
          до указвстретились
          цикл
            конст x = a[(l + r) /: 2] ;
            до найденмнш
            цикл
              если a [i] >= x то найденмнш !
              иначе i := i + 1 все
            повторить;
            до найденбольш
            цикл
              если a [j] <= x то найденбольш !
              иначе j := j — 1 все
            повторить ;
            если i <= j то % поменять местами ai и aj
              ф64 w := a [i] ;
              a [i] := a [j] ; a [j] := w ;
              i := i + 1 ; j := j — 1
            все ;
            если i > j то указвстретились ! все
          повторить;
          если l < j то раздел (l , j) все ;
          если i < r то раздел (i , r) все
          % иначе раздел состоит из одного элемента,
          % который находится на своем месте
        конец;          % раздел
      раздел (0 , длина a — 1)
    конец ;             % быстрсорт
  b <:= ф64 мконст ф64 (20 , 21 , 16 , 18 , 16) ;
  запф (печ , :'' исходный массив:'' , b : 10g (10)) ;
  быстрсорт (b) ;
```

```
    вапф (печ , : ''результирующий массив:'' , b : 10g (10))
конец
```

Пример 11. Проблема восьми ферзей. Возвратный алгоритм.

```
начало
    ф64
        a := 0 . ([7 : 8] : 4 ''ff'') ,
        % a . [к] = истина — нет ферзя на к-й
        %вертикали, 0 <= к <= 7
        b := 0 . ([14 : 15] : 4 ''7fff'') ,
        % b . [m] = истина — нет ферзя на m-й
        %диагонали, 0 <= m <= 14
        c := 0 . ([14 : 15] : 4 ''7fff'') ;
        % c . [m] = истина — нет ферзя на m-й обратной
        % диагонали
    конст
        x = лок вект [8] ф4 ,
        % x [i] = к положение i-го ферзя на горизонтали
        печ = позп (ацпу) ;
    текст
        поленебьется (i1 , j1) =
            a . [j1] и b . [i1 + j1] и c . [i1 — j1 + 7] ,
        полесвободно (i1, j1 , своб) =
            a . [j1] := b . [i1 + j1] := c . [i1 — j1 + 7] := своб,
        занятьполе (i1 , j1) =
            (x [i1] := j1 ; полесвободно (i1 , j1 , ложь)) ;
    процедура можноустановить = функция (i)
        % значением функции является значение
        % параметра структурного перехода
        до успех , крайдоски
        начало % попытка установить на i-ю горизонталь
            для j до 7
            цикл
                если поленебьется (i , j) то
                    занятьполе (i , j) ;
                    если i = 7 то успех ! (истина)
                    инес можноустановить (i + 1) то успех ! (истина)
                    иначе полесвободно (i , j , истина
                    все
                все
            повторить ;
            крайдоски ! (ложь)
        конец ; % можноустановить
```

```
    если можноустановить (0) то
        запф(печ , : ''решение:'' l , x : 2xd)
    иначе
        запф(печ , : ''нет решения'')
    все
конец
```

Пример 12. Задача путешественника, см. п. 5.2 гл. 1. В программе введена оптимизация: если при исключении i-го предмета оказывается, что результат, полученный на предыдущих этапах, нельзя улучшить (даже если добавить все предметы, следующие за i-м), то очередной этап прекращается.

```
начало
    конст
        n = 6 ,
        печ = позп (ацпу) ,
        вес = лок вект [n] ф64 ,
        цен = лок вект [n] ф64 ,
        реш = лок вект [n] ф1 ,
        оптим = лок вект [n]ф1 ,
        этапзакончен = ситуация ( ) ;
    ф64 предвес := 0 , полнцен := 0 , максцен ;
    процедура рассмотреть = проц (i , теквес , максвозмцен)
        начало
            если i >= n то % все предметы рассмотрены
                % выбрать лучший набор:
                если максвозмцен > максцен то
                    оптим <:= реш ;
                    максцен := максвозмцен
                все ;
                этапзакончен !
            иначе
                если теквес + вес [i] <= предвес то
                                % добавить i-й предмет
                    реш [i] := истина ;
                    до * этапзакончен
                    (рассмотреть (i + 1 , теквес + вес [i] ,
                    максвозмцен)
                    при этапзакончен : реш [i] := ложь
                    всесит
                все
            все ;
            % попытка составить набор, не включающий
            % i-й предмет
```

```
        если (максвозмцен := максвозмцен — цен [i])
        >максцен то
            % имеет смысл :
            рассмотреть (i + 1 , теквес , максвозмцен)
        иначе
            % не имеет смысла. так как текущий набор таков,
            % что нельзя получить решение лучше оптим.
              этапзакончен !
        все
    конец; % рассмотреть
  вес <:= ф64 мконст ф64 (10 , 20 , 15 , 13 , 17 , 22) ;
  цен <:= ф64 мконст ф64 (3 , 2 , 1 , 5 , 4 , 7) ;
  для i до n — 1
  цикл полнцен := полнцен + цен [i] повторить ;
  запф (печ , : ''вес'' 6x , вес : 3zdx , : l ''ценность'' ,
  цен : 3zdx) ;
  от 1 до 6
  цикл % 6 вариантов предельного веса
    предвес := предвес + 15 ;
    максцен := 0 ;
    реш <:= заполн ложь ;
    до *этапзакончен (рассмотреть (0 , 0 , полнцен)) ;
    запф (печ , : l , предвес : 4x3zdx) ;
    для к до n — 1
    цикл % печать * для включенных предметов
      запф (печ , если оптим [к] то ''*'' иначе '' '' все
            : 2xa2x)
    повторить
  повторить
конец
```

Пример 13. Программа пакетного задания. В примере описывается задание, выполняющее вызов программы, текст которой «вложен» в текст задания.

```
начало
  конст тест = тфайл (имяяз : ''авт'')
    *!!*
  проц (n , фа , фb)
  начало
    конст a = лок вект [n] , b = лок вект [n] ,
      печ = позп (ацпу) ;
    процедура скапр = функция (x, y)
      (ф128 сум := 0 ;
      для i до длина x — 1
```

```
        цикл % суммирование с удв. точностью :
            сум := сум + x [i] умнд y [i]
        повторить ;
            ф64окр сум) ; % округление результата
    читф (фa , a) ; читф (фb , b) ; % ввод массивов
    вапф(печ , : ''скалярное произведение='' ,
    скапр (a , b) : g (0 , 1)) % печать результата
конец % теста
!! ,
bba = тфайл
    *!!* ,
    1  2  3  4
    !! ,
bbb = тфайл
    *!!*
    5  6  7  8
    !! ;
тест (4, позп (bba) , позп (bbb))
конец % задания
```

Это же задание можно записать короче, ликвидировав описания констант и изобразив все файлы непосредственно в конструкции ⟨вызов⟩:

```
начало
    тфайл (цмяяз : ''авт'')
    *!!*
    проц (n , фa , фb)
    начало
        ...
    конец
    !!;
    (4 ,
    позп (
    тфайл
        *!!*
        1  2  3  4
        !!) ,
        позп (
        тфайл
        *!!*
        5  6  7  8
        !!))
конец
```

ГЛАВА 3

СРЕДСТВА ВЗАИМОДЕЙСТВИЯ С ВНЕШНИМИ ОБЪЕКТАМИ

В этой главе описываются средства организации обмена и архивного хранения данных на внешних устройствах.

Конструкции, связанные с обменом, по своим изобразительным средствам разбиваются на две категории:

1. Конструкции, обозначающие генераторы объектов и основные функциональные операции обмена*). Эти конструкции имеют нестандартный синтаксис вызова процедуры (см. пп. 2.1, 7.1) и начинаются с идентификаторов *чит*, *вап*, *нов*, *уст*, *читпар*, *ваппар*, *устпар*, *читатр*, *ваптр*, *венфайл*, *генконтейнер*, *файл*, *контейнер*, *позп*, *бвв*, *тбуф*, *мпр*.

2. Конструкции, обозначающие дополнительные операции. Эти конструкции имеют стандартный синтаксис вызова процедуры, например *ликвидвнеш* (*объект*). Параметры в общем случае могут иметь форму ⟨выражения⟩.

* В описании семантики операций, кроме их логического смысла, приводится системная интерпретация, т. е. описываются действия, которые фактически производит внешнее устройство. В этой связи необходимо отметить, что в случае строго вводных устройств (чтение с перфокарт, чтение с перфоленты) и строго выводных устройств возможно промежуточное хранение информации на магнитных дисках или барабанах (виртуальные файлы). Для выводных файлов это означает, что информация записывается в виртуальный файл и лишь впоследствии целиком выводится на реальное внешнее устройство. Для вводных файлов это означает, что информация сначала целиком передается в виртуальный файл, а программа затем счи-

*) Под термином «функциональные операции» здесь понимаются операции, реализованные в виде процедур операционной системы. В дальнейшем слово «функциональные» будет для краткости опускаться.

тывает из этого файла. Системная интерпретация, приведенная в семантике, не учитывает возможности промежуточного хранения, а описывает гипотетическую картину непосредственного общения программы с внешним устройством. *

1. Объекты. Общие сведения

1.1. Компоненты внешних объектов. Внешними объектами являются объекты, располагающиеся не в оперативной памяти, а на внешних носителях, таких, как магнитные диски, магнитные ленты, перфокарты и пр. В процессе выполнения задача может создавать или использовать внешние объекты, существовавшие до начала ее выполнения, ликвидировать внешние объекты или оставлять их после завершения работы.

Основным представителем внешних объектов является файл. Файл состоит из двух основных компонент — заголовка файла и его содержимого — последовательности элементов файла. Доступ к содержимому осуществляется косвенно посредством заголовка файла. Кроме того, в заголовке файла хранятся его атрибуты, т. е. ряд физических и логических характеристик файла.

Содержимое файла может иметь некоторую логическую структуру. Например, это может быть последовательность строк текстовой информации или объектный код программы. Другим примером является случай, когда содержимое файла представляет собой модель некоторого объекта, например, файла. Такие модели называются псевдообъектами. Примером псевдофайла является модель ацпу-файла на барабане или диске (псевдо-ацпу).

Под термином «внешние объекты» подразумеваются не только файлы в традиционном смысле, но и такие образования, как псевдообъекты, справочники и контейнеры. Контейнер — это один или несколько пакетов магнитных дисков или бобин магнитных лент, содержащих один или несколько файлов.

В последующем описании различаются следующие разновидности внешних объектов:

1. *Файлы на магнитном барабане* (*мб-файлы*). Компонентами мб-файла являются заголовок, последовательность элементов и справочник внешних связей, содержащий ссылки из данного файла на другие объекты.

2. *Файлы на магнитных дисках* (*мд-файлы*). Компонентами мд-файла являются заголовок, последовательность элементов и справочник внешних связей. Каждый мд-файл в свою очередь является компонентой некоторого контейнера на магнитных дисках.

3. *Контейнеры на магнитных дисках (мд-контейнеры).* Компонентами мд-контейнера являются заголовок, мд-файлы, базовый справочник контейнера и другие справочники, расположенные в данном контейнере.

4. *Файлы на магнитных лентах (мл-файлы).* Компонентами мл-файла являются заголовок и последовательность элементов. Мл-файл может располагаться на одной или нескольких бобинах магнитных лент. Каждый мл-файл в свою очередь является компонентой некоторого контейнера на магнитных лентах.

5. *Контейнеры на магнитных лентах (мл-контейнеры).* Компонентами мл-контейнера являются заголовок и мл-файлы, расположенные в данном контейнере.

6. *Файлы на прочих носителях,* а именно на перфокартах (при вводе это чпк-файлы, а при выводе — зпк-файлы), на бумажном носителе (ацпу-файлы), на перфоленте (при вводе это чпл-файлы, а при выводе — зпл-файлы), на экране (ацд-файлы), на бумажном носителе пишущей машинки (пм-файлы). Компонентами этих файлов являются заголовок и последовательность элементов.

7. *Справочники.* Компонентами справочника являются элементы справочника. Элемент справочника может содержать ссылку на некоторый внешний объект, ссылку на элемент справочника или пустую ссылку. С элементом справочника связано алфавитно-цифровое обозначение (имя) этого элемента.

8. *Программы.* Программа является псевдообъектом, содержащимся в файле объектного кода. Программа рассматривается как объект, к которому применимо только понятие выполнения.

9. *Псевдофайлы.* Как и в случае обычного файла, компонентами псевдофайла являются заголовок и последовательность элементов.

Мд- и мл-контейнер состоит из последовательности пронумерованных математических томов. Математическому тому может быть поставлен в соответствие физический том. Том — это пакет дисков в случае мд-контейнера или бобина магнитной ленты в случае мл-контейнера. На конкретной установке машины каждый физический том имеет уникальный регистрационный номер. Регистрационный номер назначается при разметке тома, наносится на его внешней поверхности и записывается в метку тома.

1.2. Доступ к объектам. Представителем внешнего объекта в оперативной памяти является оперативный объект связи с внешним объектом. Такими объектами являются: объект связи с файлом, объект связи с контейнером и указатель внешнего объекта.

Связь с файлом. Для каждой разновидности обмена существует свой объект связи с файлом. Для синхронного непосредственного обмена (т. е. обмена «программный массив ↔ файл» синхронно с выполнением программы) используется заголовок открытого файла. Заголовок открытого файла представляет собой копию заголовка файла в оперативной памяти. Для параллельного непосредственного обмена (т. е. обмена «программный массив ↔ файл» параллельно с выполнением программы) используется блок ввода/вывода (БВВ). БВВ в свою очередь связан с заголовком открытого файла. Для однобуферного обмена (т. е. обмена «один рабочий буфер ↔ файл») используется позиционная переменная. Позиционная переменная в свою очередь связана с заголовком открытого файла. Для многобуферного обмена (т. е. обмена «несколько рабочих буферов ↔ файл») используется таблица буферов. Таблица буферов в свою очередь связана с заголовком открытого файла.

Для того чтобы начать обмен, задача должна открыть файл, т. е. создать в оперативной памяти соответствующий объект связи с файлом.

Связь с контейнером. В ряде операций над контейнером требуется, чтобы контейнер был в открытом состоянии. Доступ к открытому контейнеру осуществляется косвенно посредством заголовка открытого контейнера. Заголовок открытого контейнера представляет собой копию заголовка контейнера в оперативной памяти.

Указатель внешнего объекта — это оперативный объект связи, который содержит ссылку на некоторый внешний объект или элемент справочника. Будем говорить, что указатель установлен на данный объект или элемент. С помощью соответствующих операций указатель можно установить на другой внешний объект или элемент справочника. Причем, если указатель устанавливается на файл или контейнер, то это не приводит к открытию последних.

Доступ к справочнику осуществляется с помощью указателя внешнего объекта, установленного на данный справочник. Доступ к элементу справочника осуществляется с помощью указателя внешнего объекта, установленного на данный элемент справочника.

Доступ к объекту связи осуществляется через посредство указателя объекта связи. Таким образом, когда программа выполняет действия над внешними объектами, то первичным объектом, обеспечивающим доступ к внешнему объекту, является указатель объекта связи: этот указатель (а не объект связи и не внешний объект) является значением соот-

ветствующих ⟨выражений⟩, значением программных переменных и констант.

Этот факт будет в дальнейшем неявно подразумеваться, но для сокращения изложения там, где это возможно, вместо термина «указатель объекта связи» используется название самого объекта связи. Например, вместо термина «указатель заголовка открытого файла» используется термин «заголовок открытого файла». Ниже в этом пункте вводятся дальнейшие соглашения о сокращении терминов, связанные с косвенным доступом к объектам.

* *Открытие программы.* Открыть можно не только файл или контейнер, но также и программу. В простейшем случае в результате открытия программы выдается процедура. Выполнение программы заключается в вызове этой процедуры.

В общем случае в тексте программы можно описать те действия, которые выполняются при ее открытии, и тот объект, который является результатом открытия. В частности, результатом открытия может быть некоторый внешний объект, или элемент справочника. Таким образом, программу также можно рассматривать как один из возможных путей доступа к внешнему объекту или элементу справочника. *

Косвенный доступ к внешнему объекту. Для всех операций, работающих с внешним объектом как с единым целым, несущественно, какой оперативный объект используется для связи. Эти операции минуют все промежуточные звенья и доходят до целевого объекта. Кроме оперативного объекта, промежуточным звеном может являться:

1. *Элемент справочника.* В этом случае в качестве следующего звена рассматривается объект или элемент справочника, на который ссылается исходный.

* 2. *Программа.* Если этот объект не является целевым, то он рассматривается как промежуточное звено. В этом случае программа открывается и результат открытия рассматривается в качестве следующего звена.

3. *Файл объектного кода.* Если целевой объект — не файл, то в качестве следующего звена рассматривается программа, содержащаяся в этом файле. *

В дальнейшем там, где возможно, вместо названия объекта связи, обеспечивающего доступ к внешнему объекту, и вместо цепочки «указатель внешнего объекта → элемент справочника → ... → целевой внешний объект» употребляется название целевого внешнего объекта. Например, ряд конструкций может оперировать как с открытым файлом, так и с закрытым. В этом случае вместо терминов «заголовок открытого файла», «позиционная переменная», «указатель, установленный на файл», «указатель, установленный на элемент справочника, ссылающийся на

файл» и т. д. используется термин «файл». Кроме того, имея в виду возможный неявный проход по промежуточным звеньям, будем говорить, что указатель внешнего объекта приводит к целевому объекту.

* Эти же соглашения действуют и тогда, когда промежуточным звеном цепочки является файл объектного кода или программа. *

Термин «тип внешнего устройства» обозначает тип устройства, на котором может располагаться внешний объект.

1.3. Создание внешних объектов. Внешний объект создается с помощью ⟨генератора внешнего объекта⟩, например:

*мбф*1 := *генфайл* (*барабан*)
мбфайл := *генфайл* (*барабан*)

здесь создаются два мб-файла (с атрибутами, выбираемыми по умолчанию);

*диск*1 := *генконтейнер* (*мдконт*)

здесь создается мд-контейнер (с атрибутами, выбираемыми по умолчанию);

*мдф*1 := *генфайл* (*диск*1)

здесь создаются мд-файл (с атрибутами, выбираемыми по умолчанию) в контейнере *диск*1;

*спр*1 := *генспр* (*барабан*)

здесь создается мб-справочник.

В результате выполнения ⟨генератора внешнего объекта⟩ объект создается, но не открывается. Кроме внешнего объекта, создается указатель внешнего объекта. Этот объект связи устанавливается на созданный внешний объект, и указатель объекта связи выдается в качестве результата генерации. В приведенных выше примерах указатель объекта связи присваивается переменным *мбф*1, *мбфайл*, *диск*1, *мдф*1, *спр*1 (предполагается, что формат этих переменных — ф64).

1.4. Создание объектов связи. Оперативные объекты создаются с помощью ⟨генератора объекта связи⟩, например:

*мбф*1*нпс* := *файл* (*мбф*1)

здесь открывается файл *мбф*1, т. е. создается заголовок открытого файла. С помощью этого заголовка можно в дальнейшем осуществлять непосредственный обмен с файлом:

зап (*мбф*1*нпс*, *адреснач*1, *массив*1)

здесь в файл, начиная с адреса *адреснач*1, записывается массив *массив*1;

*мдф*1*буф* := *позп*(*мдф*1)

здесь, во-первых, открывается файл *мдф*1, во-вторых, создается позиционная переменная для буферизованного обмена с этим файлом. Позиционная переменная (точнее,— ее указатель, см. п. 1.2) становится значением переменной *мдф*1*буф*. С помощью этой позиционной переменной можно осуществлять однобуферный способ буферизованного обмена с данным файлом (см. § 10).

* Ниже приведен пример, демонстрирующий создание указателя внешнего объекта. Этот указатель является подвижным, т. е. его можно устанавливать на различные внешние объекты:

*ув*1 := *генув*() *

1.5. Внешний контекст. Элемент справочника содержит ссылку на внешний объект. Этот внешний объект является файлом, контейнером, справочником. Последний, в свою очередь, может содержать ссылки на внешние объекты. Совокупность, состоящая из справочника А и внешних объектов, прямо или косвенно (через посредство промежуточных справочников) достижимых из этого справочника, называется внешним контекстом. Справочник А называется корневым справочником этого внешнего контекста.

Внешний контекст пользователя — это внешний контекст, состоящий из объектов, к которым данный пользователь может иметь доступ. Корневой справочник внешнего контекста пользователя создается при регистрации этого пользователя в системе. Первоначально корневой справочник состоит из одного элемента. Этот элемент содержит ссылку на корневой справочник глобального внешнего контекста, доступного всем пользователям.

К корневому справочнику применимы все операции над справочниками (создание и уничтожение элементов, поиск элемента по его имени, запись в элемент ссылки на объект). Из программы пакетного задания корневой справочник пользователя доступен с помощью идентификатора *свойвк* (*свой внешний контекст*).

Внешний контекст программы. В тексте программы могут встречаться алфавитно-цифровые внешние имена, предназначенные для упоминания внешних объектов. Соответствующие объекты ищутся (по их именам) во внешнем контексте данной программы. Каждая программа может, вообще

говоря, иметь свой собственный внешний контекст или контекст, который частично или полностью пересекается с контекстом других программ.

Внешний контекст задания. Пакетное задание — это программа, внешним контекстом которой является внешний контекст пользователя, составившего задание.

1.6. Операции над справочником. К числу основных операций над справочниками относятся создание элемента, уничтожение элемента и запись в элемент ссылки на объект или элемент другого справочника.

Создание элемента. Справочник можно расширять, создавая в нем новые элементы, например:

создэлспр (*спр*1, "*файл*3", *мбфайл*)

Здесь в справочнике *спр*1 создается элемент с обозначением "*файл*3" и в него знаносится ссылка на файл *мбфайл*.

Эта же операция используется для создания элементов в корневом справочнике внешнего контекста программы (последний обозначается идентификатором *свойвк*). Например:

создэлспр (*свойвк*, "*файл*2", *мбфайл*)

К элементу, созданному в последнем примере, можно обращаться по внешнему имени //*файл*2 (см. ниже).

Запись ссылки. Содержимое элемента справочника можно изменять:

запспр (//*файл*2, *генфайл* (*барабан*))

Здесь в элемент "*файл*2" заносится ссылка на вновь созданный мб-файл, а старая ссылка уничтожается.

Уничтожение элемента. Справочник можно сокращать, уничтожая его элементы, например:

ликвидвнеш (//*файл*2)

Здесь элемент "файл2" ликвидируется. Объект, на который ссылается элемент, также ликвидируется, если на него нет других ссылок.

1.7. Именование внешних объектов. Простейшая форма конструкции ⟨внешнее имя⟩ предназначена для обозначения указателя, установленного на элемент какого-либо справочника из внешнего контекста программы. Например, //*файл*2 — это указатель, приводящий к объекту, ссылка на который содержится в элементе "*файл*2" корневого справочника внешнего контекста

программы. Если объект, к которому приводит указатель — это файл, то его открытие можно записать следующим образом:

*ф*1 ::= *файл*(//*файл*2).

Другой пример, если в элементе "*пак*1" справочника содержится ссылка на мд-контейнер, то создание в нем файла можно записать следующим образом:

*мдф*1 ::= *генфайл*(//*пак*1)

Внешнее имя может быть многослоговым, например:

//*спа*//*спв*//*х*

Здесь элемент "*спа*" должен ссылаться на справочник, в котором элемент "*спв*" в свою очередь ссылается на справочник, в котором элемент "*х*" ссылается на некоторый внешний объект. В целом данное многослоговое внешнее имя обозначает указатель, приводящий к этому последнему внешнему объекту.

Указатель, приводящий к объекту из внешнего контекста программы, можно пересылать без ограничений. Например, его можно передавать в качестве параметра в другую программу, возможно, имеющую другой внешний контекст. При этом контекстная информация не теряется.

1.8. Именование объектов на съемных носителях. Доступ к объекту на съемном носителе (диски, ленты, перфокарты и пр.) можно организовать двумя способами:

1. Можно ссылку на объект занести в элемент какого-либо справочника (в частности,— в элемент корневого справочника внешнего контекста программы), и затем использовать обозначение элемента, как указано выше.

2. Можно обращаться к объекту с помощью его собственной метки, точнее — алфавитно-цифрового имени, находящего в метке файла или контейнера (см. атрибуты *имяфайла, имяконтейнера*). Например, внешнее имя мл-контейнера с меткой "*арх*1" записывается следующим образом:

ксн//*арх*1

Здесь *ксн* — стандартный идентификатор, с которого начинаются внешние имена объектов на съемных носителях.

1.9. Ссылка на объект. Для реализации связи задачи с объектом и для связи одного объекта с другим система создает скрытые ссылки соответственно из задачи на объект и из объекта на объект. Как правило, эти ссылки в явном виде недоступны программе. Для выполнения операций над объектом программы используется указатель объекта. Этот указатель

является доступным программе эквивалентном скрытой ссылки. Например, при открытии файла:

*мбф*1 := *мбф*2 := *файл*(//*файл*2)

в документации о задаче появляется скрытая ссылка на открытый файл, а значением двух переменных *мбф*1 и *мбф*2 становится указатель заголовка открытого файла. При этом регистрируется наличие одной скрытой ссылки (а не двух ссылок) из задачи на файл. Если присваивать этот указатель другим переменным, передавать его как параметр и пр., то эти действия не увеличивают количество скрытых ссылок из задачи на файл.

Прикрепление объекта и время жизни объекта. Прикрепление объекта В к другому объекту А состоит в том, что создается скрытая ссылка из А на В. Аналогично, прикрепление объекта В к задаче А состоит в том, что создается скрытая ссылка из задачи А на объект В. Вообще говоря, одновременно могут существовать несколько скрытых ссылок на данный объект из нескольких задач и/или объектов, или даже несколько скрытых ссылок из одной задачи и/или одного внешнего объекта.

Прикрепление происходит при создании нового объекта, при открытии внешнего объекта, при связывании двух объектов.

Откреплением объекта В от объекта А состоит в том, что ликвидируется скрытая ссылка из А на В. Аналогично открепление объекта В от задачи А состоит в том, что ликвидируется ссылка из А на В. Если В прикреплен к задаче А с помощью нескольких ссылок, то количество этих ссылок уменьшается на 1.

После очереднего открепления объекта может оказаться, что этот объект не прикреплен ни к одному объекту и ни к одной задаче. В этом случае от объекта открепляются все прикрепленные к нему, и объект ликвидируется. Таким образом, время жизни объекта определяется наличием скрытых ссылок на него из задачи и/или из других объектов.

Открепление объекта происходит при выполнении ряда операций над объектами связи, при выполнении операций явного открепления, по окончании выполнения задачи, к которой объект прикреплен. Все эти случаи открепления указаны в соответствующих разделах описания. Кроме того, как указано выше, открепление объекта В от А может происходить рекурсивно, как результат открепления объекта А от другого объекта или задачи.

1.10. Ликвидация объектов. Ликвидация оперативного объекта означает освобождение оперативной памяти, занимаемой этим объектом.

Ликвидация внешнего объекта означает в основном освобождение памяти, занимаемой этим объектом на внешнем носителе: освобождение памяти на барабане или внутри мд-контейнера, освобождение мд-контейнера, освобождение памяти в мл-контейнере, освобождение всех лент, занятых под мл-контейнер.

Ликвидация объекта производится либо неявно, если не осталось скрытых ссылок на объект (см. выше), либо явно с помощью процедуры *ликвидвнеш*. *

2. Создание, открытие и ликвидация объектов

2.1. Генератор объекта. Описываемые ниже генераторы предназначены для создания внешних объектов и оперативных объектов связи.

Внешние объекты.

⟨генератор внешнего объекта⟩ ::=
⟨спецификация внешнего⟩({⟨выражение⟩,}⟨выражение⟩
{,⟨список установок атрибутов⟩})
⟨спецификация внешнего⟩ ::= *генфайл* | *генконтейнер* |
генспр

⟨Спецификация внешнего⟩ определяет тип создаваемого объекта, а ⟨список установок атрибутов⟩ — его атрибуты. Значение второго ⟨выражения⟩ задает принадлежность создаваемого объекта. Этот параметр называется основой генератора. Его смысл зависит от типа создаваемого объекта.

В случае спецификации *генконтейнер* основой генератора должна быть процедура создания контейнера (см. п.2.2). С помощью этой процедуры создается контейнер. В случае мд-контейнера в нем создается базовый справочник данного контейнера.

Пример 1. Создание контейнера, базовым томом которого является том с регистрационным системным номером МД100.

*дискпак*1 := *генконтейнер*(*мдконт*, *нмртома*: "*мд*100")

В случае спецификации *генфайл* основой генератора может быть:

1. Процедура создания файла (см. п. 2.2). С помощью этой процедуры создается файл.

2. Процедура создания контейнера (см. п. 2.2). В этом случае создается контейнер, и в нем — файл.

3. Контейнер. В этом случае создается файл в данном контейнере.

Пример 2. Создание файла в контейнере *дискпак*1 (см. пример 1) с установкой атрибутов.

*мдф*1 := *генфайл*(*дискпак*1, *длинлиста*:1024,
максдлинблока:256)

* В случае спецификации *генспр* создается новый справочник. Основой может быть:

1. Процедура создания объекта на барабане (стандартная процедура *барабан*). В этом случае создается мб-справочник.

2. Мд-контейнер. В этом случае справочник создается в данном контейнере.

Используя первое ⟨выражение⟩, в конструкцию ⟨генератор внешнего объекта⟩ можно подать подвижный указатель. Этот указатель установится на вновь созданный объект. В противном случае указатель создается неявно и локализуется в ближайшем блоке. В обоих случаях указатель устанавливается на вновь созданный объект, объект прикрепляется к указателю, и указатель выдается в качестве результата выполнения конструкции. Если файл создается в контейнере, то последний прикрепляется к файлу. *

Объекты связи.

⟨генератор объекта связи⟩ ::=
⟨спецификация объекта связи⟩
({⟨выражение⟩,}⟨список установок атрибутов⟩)
|⟨спецификация объекта связи⟩({⟨список установок атрибутов⟩})
⟨спецификация объекта связи⟩ ::=
файл | *позп* | *тбуф* | *бвв* | *контейнер* | *прогр* | *мпр* | *генув*

⟨Спецификация объекта связи⟩ определяет тип создаваемого объекта, а ⟨список установок атрибутов⟩ — его атрибуты. Значение ⟨выражения⟩ задает основу генератора. В случае спецификации *файл*, *бвв*, *позп*, *тбуф* основой генератора может быть файл (в этом случае устанавливается связь с данным файлом) или объекты, перечисленные выше для спецификации *генфайл* (в этом случае создается новый файл).

В зависимости от спецификации объекта производятся следующие действия:

файл — открывается файл. Это означает, что если в оперативной памяти еще нет копии заголовка данного файла, то она создается. В противном случае используется уже существующий заголовок открытого файла;

позп — файл открывается и для него создается позиционная переменная.

* *тбуф* — открывается таблица буферов. Это означает, что если в оперативной памяти еще нет таблицы буферов для данного файла, то файл открывается и для него создается таблица буферов. В противном случае используется уже существующая таблица буферов;

бвв — файл открывается и для него создается блок ввода/вывода;

генув — создается подвижный указатель внешнего объекта. Первоначально указатель содержит пустую ссылку. Основа генератора в данном случае опускается.

При создании заголовка открытого файла (позиционной переменной, таблицы буферов, блока ввода/вывода) происходит прикрепление файла (заголовка открытого файла) к созданному объекту связи.

В генераторе блока ввода/вывода можно опускать основу. В этом случае создается блок ввода/вывода, не связанный с

каким-либо файлом. К этому блоку можно в дальнейшем прикрепить заголовок открытого файла. Это делается путем модификации атрибута *прикрепфайл*.

Массив процедур реакций. Спецификация *мпр* указывает на то, что создается массив процедур реакций на ошибки обмена (см. гл. 5). В этом случае основа генератора опускается. *

Открытие контейнера. В случае спецификации *контейнер* основой генератора может быть контейнер или процедура создания контейнера (в этом случае создается новый контейнер). Контейнер открывается; открытие происходит аналогично открытию файла, но с учетом того, что в данном случае объектом связи является заголовок открытого контейнера.

Результат. В перечисленных выше случаях значением конструкции является созданный объект связи.

* Открытие программы. Если указана спецификация *прогр*, то основой генератора может быть:

1. Файл объектного кода программы (см. атрибут *типфайла*). В этом случае открывается содержащаяся в нем программа (см. § 12 гл. 2)

2. Текстовый файл (см. атрибут *типфайла*), содержащий текст программы на некотором языке программирования. Если справочник внешних связей (СВС) этого файла еще не содержит ссылку на файл объектного кода программы, то текст программы транслируется соответствующим транслятором (см. атрибут *имяяз*), ссылка на полученный файл кода заносится в СВС исходного файла (см. атрибут *кодпрогр*) и программа открывается. В противном случае программа открывается по уже имеющейся ссылке.

Внешним контекстом программы, полученной в результате неявной трансляции, становится внешний контекст исходного текстового файла (см. атрибут *внешконт* и § 12).

3. Программа. В этом случае данная программа открывается. *

Текущая позиция. Считается, что в процессе обмена со строго последовательными файлами (мл-, пк-, ацпу-, пл-, ацд-, пм-файлы) существует текущая позиция. Текущая позиция — это положение головки записи/считывания по отношению к началу файла. Аналогичное понятие существует и для мл-контейнера (подробнее см. § 6).

Непосредственно после того, как файл (контейнер) создан, текущая позиция установлена на начало файла (контейнера), т. е. в положение, предшествующее первому элементу файла (первому файлу контейнера). В процессе обмена текущая позиция перемещается по файлу (контейнеру), см. § 8.

Текущая позиция может быть установлена на конец файла (контейнера); такая позиция есть положение, следующее за последним элементом файла (последним файлом контейнера).

При открытии существующего или вновь созданного строго последовательного файла текущая позиция устанавливается на начало файла (относительно мл-файла и мл-контейнера см. дополнительно § 6).

Понятие «текущая позиция» используется и при описании буферизованного обмена с файлами, расположенными на устройствах любого типа. В процессе однобуферного обмена информация о текущей позиции хранится в позиционной переменной (см. § 10). Считается, что при создании позиционной переменной текущая позиция установлена на начало файла (исключение составляет случай мл-файла: вновь созданная позиционная переменная в начальный момент отображает фактическое текущее положение головки по отношению к текущей ленте файла). В последующем позиционная переменная отслеживает перемещение текущей позиции.

* *Локализация объектов.* Если в генераторе объекта атрибуту *локал* присваивается значение **истина**, то объект связи (указатель, заголовок открытого файла и пр.) прикрепляется к ближайшему блоку задачи. Если же атрибуту присваивается значение **ложь**, то объект прикрепляется к задаче. По умолчанию выбирается значение **истина**.

Локальный оперативный объект, прикрепленный к блоку А, может быть явно откреплен от него с помощью вызова процедуры *открел* внутри блока А или внутри блока, динамически или статически вложенного в А. Если явного открепления нет, то локальный объект неявно открепляется от блока, к которому он прикреплен, по окончании выполнения этого блока. В этом случае над объектом выполняются те же действия, что и при явном откреплении.

Глобальный оперативный объект может быть явно откреплен от задачи, к которой он прикреплен, с помощью вызова в этой задаче процедуры *открел*.

Если к моменту окончания задачи оказывается, что остались глобальные ссылки из этой задачи на объект, то этот объект неявно открепляется от задачи с уничтожением всех ссылок из задачи на него.

Оперативный объект можно использовать до тех пор, пока он прикреплен к блоку, задаче или другому объекту. *

2.2. Процедуры создания объектов. В общем случае процедура создания объекта может быть определена пользователем. В то же время существует ряд системных стандартных процедур создания объектов.

В пп. 1)—5) ниже приведены идентификаторы, используемые как средство обозначения процедур генерации внешних объектов. Процедуры генерации являются атрибутами задачи. В общем случае их можно указывать при запуске задачи, что обеспечивает виртуальное использование внешних устройств. По умолчанию значением атрибутов являются стандартные

процедуры, семантика которых описывается ниже. Пп. 6) и 7) касаются тех случаев, когда значением основы генератора является контейнер.

1) *барабан.* Тип внешнего устройства — магнитный барабан. Создается мб-файл.

2) *диск.* Тип внешнего устройства — магнитный диск. Создается мд-файл в стандартном системном мд-контейнере.

3) *мдконт.* Тип внешнего устройства — магнитный диск. Создается мд-контейнер.

Если в генераторе задан атрибут *нмртома*, то базовый пакет контейнера создается на томе с заданным регистрационным номером. В противном случае базовый пакет создается на каком-либо свободном томе. Дальнейшее назначение физических томов математическим происходит автоматически при исчерпании свободной памяти контейнера. Кроме того, существуют средства явного назначения (см. § 6).

Если в генераторе контейнера задан атрибут *имяконт* (алфавитно-цифровое имя в метке контейнера), то мд-контейнер прикрепляется к контексту съемных носителей. В дальнейшем к контейнеру можно обращаться по внешнему имени *ксн//s*, где *s* — содержимое этого атрибута (см. п. 5.4). В противном случае контейнер является временным и будет ликвидирован по окончании задачи.

4) *млконт.* Тип внешнего устройства — магнитная лента. Создается мл-контейнер.

Средства назначения физических томов математическим такие же, как и в случае мд-контейнера.

Если в генераторе объекта указан атрибут *имяконт* (алфавитно-цифровое имя в метке томов контейнера), то мл-контейнер прикрепляется к контексту съемных носителей. Если значением атрибута является строка или набор нулевой длины (пустая строка), то контейнер называется непомеченным, и обращаться к нему в дальнейшем нужно с помощью стандартного указателя *нпо*. В противном случае внешнее имя созданного контейнера — *ксн//s*, где *s* — содержимое атрибута *имя конт* (см. п. 5.4).

Если атрибут *имяконт* не задан, то созданный контейнер является временным, и будет ликвидирован по окончании задачи.

5) *впк, ацпу, впл.* Тип внешнего устройства — вывод на перфокарты, алфавитно-цифровое печатающее устройство, вывод на перфоленту. Создается впк-, ацпу-, впл-файл. Файл прикрепляется к контексту съемных носителей. Для впк- и ацпу-файлов атрибут *имяфайла* задает алфавитно-цифровое имя в

метке файла. Если в генераторе такого файла атрибут *имяфайла* не указан, то по умолчанию он инициализуется пустой строкой (непомеченный файл). Пл-файл всегда является непомеченным.

Последующее обращение к помеченному пк-файлу (для доступа по чтению) осуществляется с помощью внешнего имени *ксн//s*, где *s* — содержимое атрибута *имяфайла*. К непомеченному пк- или пл-файлу можно обращаться с помощью стандартного указателя *нпо*.

6) мд-контейнер. Создается мд-файл внутри данного мд-контейнера.

Если создание нового файла в контейнере требует превышения пределов, установленных значением атрибута *максдлинконт*, то возникает ошибка «физический конец контейнера».

7) мл-контейнер. Создается мл-файл внутри данного контейнера. Если в генераторе объекта указан атрибут *имяфайла*, то созданный файл прикрепляется к контейнеру. В дальнейшем к файлу можно обращаться по внешнему имени *ксн//s1//s2*, где *s1* — содержимое атрибута *имяконт* (имя из метки контейнера), а *s2* — содержимое атрибута *имяфайла* (имя из метки файла).

Если значением атрибута *имяконт* мл-контейнера является строка нулевой длины или пустой набор (непомеченный контейнер), то и все файлы должны иметь аналогичное значение этого же атрибута.

Новый файл (помеченный или нет) создается на том месте, куда предварительно установлена текущая позиция (подробно см. § 6).

Если создание нового файла в контейнере требует превышения пределов, установленных значением атрибута *максдлинконт*, то возникает ошибка «физический конец контейнера».

* *Системная интерпретация.* Интерпретация действий, производимых при создании внешнего объекта, состоит в следующем:

1. мб-файл. На носителе отводится место для заголовка нового файла.

2. мд-контейнер. Занимается свободное устройство и на него монтируется том (пакет дисков) с указанным в генераторе регистрационным номером (атрибут *нмртома*). Если регистрационный номер не задан, то монтируется какой-либо свободный том. Этот том отмечается как базовый пакет контейнера. В нем отводится место для заголовка контейнера.

3. мд-файл. В контейнере, где создается файл, отводится место для его заголовка.

4. мд-контейнер. Занимается свободное устройство и на него, как и в случае мд-контейнера, монтируется начальный том

(бобина магнитной ленты). На ленту записывается начальная метка тома.

5. мл-файл. Если перед созданием нового файла внутри контейнера какой-либо файл этого контейнера был открыт, то его необходимо закрыть. Новый файл создается в том месте, куда установлена текущая позиция (подробно см. § 6). На ленту записывается начальная метка файла. Конечная метка файла записывается на ленту в случае, если после операции записи выполняется какая-либо операция над данным файлом, отличная от записи. Вновь созданный файл считается последним в контейнере.

6. зпк-файл. Занимается свободное устройство и выводится перфокарта с начальной меткой файла.

7. ацпу-файл. Занимается свободное устройство и выводится начальная метка ацпу-файла.

8. зпл-файл. Занимается свободное устройство. Пл-файлы не имеют начальных и конечных меток. *

2.3. Операции открепления и ликвидации объектов.

1) *ликвидвнеш* (*объект*). Параметр *объект* — это оперативный объект связи с внешним объектом.

В случае, если параметр отличен от указателя, установленного на элемент справочника, то операция осуществляет ликвидацию внешнего объекта.

* При ликвидации внешнего объекта от него открепляются все прикрепленные к нему внешние объекты (что в свою очередь может привести к их уничтожению). Последующая попытка обращения к ликвидированному объекту со стороны данной задачи или других задач приводит к возникновению ошибки «нет внешнего объекта». *

В случае, если параметр — это указатель, установленный на элемент справочника, то элемент ликвидируется. Как следствие, возможна ликвидация объекта, ссылка на который содержится в элементе справочника. Это происходит при условии, что данная ссылка — единственная ссылка на объект.

* Если оперативный объект не является подвижным указателем внешнего объекта, то он также ликвидируется; подвижный указатель сохраняется, но в него заносится пустая ссылка.

Операция не выдает значение.

Системная интерпретация. уничтожения внешних объектов:

— мд-контейнер. Пакеты магнитных дисков, занимаемые контейнером, помечаются, как свободные. Устройство освобождается;

— мд-файл. Освобождаются ресурсы, занимаемые файлом в содержащем его контейнере, т. е. ликвидируются все математические листы файла (см. п. 8.1). Контейнер открепляется от файла;

— мл-контейнер. Ленты контейнера помечаются как свободные и сматываются на начало. Устройство освобождается;

— мл-файл. Если ликвидируемый файл является первым файлом содержащего его контейнера, то после начальной метки

контейнера записывается конечная метка контейнера. В противном случае эта метка записывается после конечной метки файла, который непосредственно предшествует ликвидируемому. В обоих случаях это означает, что ликвидируется ссылка (если она есть) из контейнера на ликвидируемый файл и, кроме того, ликвидируются файлы контейнера, которые находятся после ликвидируемого (если такие есть). Затем контейнер открепляется от файла. Если открепление контейнера не привело к перемотке и освобождению или снятию ленты, то лента остается в позиции конечной метки контейнера и устройство не освобождается. *

2) *откреп* (*объект*). Параметр *объект* — это объект связи с файлом или контейнером. Операция осуществляет закрытие файла или контейнера.

* Закрытие состоит в том, что объект связи открепляется от соответствующего блока или от задачи — в зависимости от того, является ли объект локальным или глобальным.

Если открепление объекта привело к тому, что не осталось ссылок на данный объект, то:

— данный объект уничтожается;

— от данного объекта открепляются все прикрепленные к нему объекты (внешние или оперативные), что в свою очередь может привести к уничтожению последних.

Операция не выдает значение.

Если в результате открепления внешнего объекта ликвидированы не все ссылки из других объектов связи и/или из задач на него, то фактически с внешним объектом никаких действий не производится.

Если же ликвидированы все подобные ссылки, но остались ссылки из внешнего контекста, то с внешним объектом производятся следующие действия:

— мд-контейнер. Контейнер снимается и устройство освобождается;

— мл-контейнер. Ленты контейнера сматываются на начало и снимаются, устройство освобождается;

— мл-файл. Контейнер открепляется от файла. Если это не привело к перемотке и освобождению или снятию ленты (т. е. файл закрыт, а контейнер остался открыт), то в зависимости от атрибута *направление* лента перематывается на конец или на начало файла;

— зпк-файл. Выводится перфокарта, соответствующая конечной метке пк-файла, и устройство освобождается;

— ацпу-файл. Выводится конечная метка ацпу-файла и устройство освобождается;

— чпк-, зпл-, чпл-, ацд-, пм-файл. Устройство освобождается. *

3. Атрибуты объектов и доступ к атрибутам

Атрибуты создаваемого объекта (внешнего или объекта связи) определяются следующим образом. При создании внешнего объекта значение атрибута *тип* (*тип внешнего устройства*)

задается процедурой, осуществляющей создание. Значения остальных атрибутов определяются исходя из ⟨списка установок атрибутов⟩ (см. § 18 гл. 2) генератора. Атрибуты, не указанные в списке, полагаются равными значениям, выбираемым по умолчанию.

Семантика атрибутов описана в гл. 4. Для каждого атрибута указано, является ли он атрибутом внешнего объекта или атрибутом открытия (см. ниже). В этой же главе приведены идентификаторы атрибутов. Считается, что каждый такой идентификатор описан в стандартном контексте программы как идентификатор константы. Значениями этих констант являются номера соответствующих атрибутов.

3.1. Принадлежность атрибутов. Различаются атрибуты внешнего объекта и атрибуты открытия (или атрибуты объекта связи).

Атрибуты внешнего объекта имеют следующие особенности:

1. Значение атрибута хранится в заголовке, находящемся на носителе вместе с самим внешним объектом.

2. Если атрибут не указан в ⟨генераторе внешнего объекта⟩, то его значение берется из первого открытия. Если же он и в открытии не указан, то полагается равным значению, выбираемому по умолчанию.

3. Если в ⟨генераторе объекта связи⟩ указан уже установленный атрибут внешнего объекта, то осуществляется контроль соответствия. При несоответствии возникает «ошибка атрибута».

Атрибуты открытия характеризуют текущее открытие внешнего объекта и имеют следующие особенности:

1. Значение атрибута хранится в оперативном объекте связи. В генераторе внешнего объекта можно указать умолчание для некоторых атрибутов открытия.

2. Если атрибут не указан в генераторе оперативного объекта, то он полагается равным значению, находящемуся в заголовке соответствующего внешнего объекта. Если значение, выбираемое по умолчанию, не было задано явно, то выбирается стандартное значение.

3. Используя указатель, установленный на внешний объект, можно модифицировать находящееся в заголовке умолчание для атрибута оперативного объекта.

3.2. Доступ к атрибутам. В этом пункте описаны особенности применения конструкций ⟨модификация атрибутов⟩ и ⟨запрос атрибутов⟩ (см. § 18 гл. 2) в случае объектов связи и внешних объектов.

Модификация атрибутов. Ряд атрибутов объектов связи и внешних объектов имеют одинаковые номера; соответствующие идентификаторы констант также совпадают. Чтобы избежать двусмысленности, вводится следующее соглашение о порядке установления принадлежности атрибута. Пусть первым параметром конструкции является объект связи X и требуется модифицировать атрибут А. Сначала проверяется, есть ли у объектов такого типа, как объект X, атрибут А. Если есть, то обрабатывается атрибут А объекта X. В противном случае считается, что А — это атрибут внешнего объекта.

В частности, если требуется модифицировать умолчание для атрибутов открытия или такой атрибут внешнего объекта, для которого имеется совпадающий по номеру атрибут объекта связи, то необходимо воспользоваться указателем, установленным на нужный внешний объект, передав его в качестве первого параметра в конструкцию ⟨модификация атрибутов⟩.

* Уточнение принадлежности атрибута. Списку установок атрибутов может предшествовать ⟨уточнение⟩. Значение ⟨уточнения⟩ имеет следующий смысл:

— если объект является файлом, а уточнение — целым, то изменяются атрибуты листа памяти файла (для мб- и мд-файлов), причем целое определяет математический номер листа, атрибуты которого изменяются;

— если объект является контейнером, а уточнение — целым, то изменяются атрибуты мд-пакета (ленты мл-контейнера) рассматриваемого контейнера, причем целое определяет математический номер пакета (номер ленты относительно начала контейнера), атрибуты которого изменяются;

— если объект является файлом, а уточнение — указателем буфера, то изменяются атрибуты буфера, связанного с файлом. Указатель должен частично или полностью описывать буферный массив какого-либо буфера данного файла. Такой метод используется только при многобуферном способе буферизованного обмена. При однобуферном способе атрибуты текущего буфера изменяются через позиционную переменную без использования ⟨уточнения⟩. *

Пример 1. Изменение умолчания для атрибута открытия *формэлем* у файла *мдф1* (см. пример в п. 2.1).

```
запатр (мдф1, формэлем: 3)
% значение умолчания для открытия атрибута формэлем
% становится равным числу 3 (литерный формат)
```

* Пример 2. Распределение (явно заданное) физического листа для математического листа с номером, равным значению *теклист*. Лист отводится на диске класса 1.

```
запатр (мдф1, теклист, распределен : истина, класслиста : 1)
```

При модификации строковых атрибутов (*имяфайла*, *имяконт*, *сообщение*) значение второго ⟨выражения⟩ в

⟨списке установок атрибутов⟩ — это набор или указатель литерной строки. В обоих случаях заданные таким образом литеры становятся содержимым атрибута. *

Запрос атрибутов. Принадлежность запрашиваемого атрибута устанавливается так же, как и при модификации атрибута.

При запросе строковых атрибутов ⟨уточнение⟩ имеет особый смысл. Оно должно быть указано обязательно, и его значением должен являться указатель литерного с-вектора. Содержимое атрибута переписывается в этот вектор, и в качестве результата выдается указатель, описывающий ровно столько начальных элементов исходного вектора, какова длина строки, содержащейся в атрибуте. Если заданный вектор короче строки, то последняя обрезается.

Пример 1.

читатр (*ацпуф*, *страница*)
% номер текущей страницы файла *ацпуф*

Пример 2.

читатр (*мдф1*, *теклист*, *распределен*)
% выдается **истина**, если математический лист с номером
% равным значению *теклист* уже распределен. *

4. Указатель внешнего объекта

Различаются две разновидности указателей внешнего объекта:

1. Постоянный указатель (указатель-константа). Такой указатель создается конструкциями ⟨внешнее имя⟩, ⟨изображение файла⟩ и др. Существует ряд стандартных указателей такой разновидности (см. ниже).

* Постоянный указатель не содержит ссылку в форме плана поиска (см. п. 4.1). Постоянный указатель нельзя изменять с помощью операций установки указателя.

2. Подвижный указатель. Такой указатель создается генератором со спецификацией *гёнув*. С помощью соответствующих операций подвижный указатель может быть установлен на тот или иной внешний объект (файл, псевдообъект, контейнер. справочник), на элемент некоторого справочника, или может содержать пустую ссылку. Ссылка в подвижном указателе содержится в форме физической ссылки (см. п. 4.1). Установка подвижного указателя на закрытый файл не приводит к открытию файла. Установка указателя на закрытый контейнер приводит к неявному открытию последнего. *

Считается, что указатель, установленный на мд-контейнер, одновременно установлен и на базовый справочник этого контейнера.

Стандартные постоянные указатели:

1) *свойкод, свойвк* — идентификаторы констант, описанных по умолчанию в собственном контексте каждой программы (см. § 12 гл. 2). Значение константы *свойкод* — это постоянный указатель, приводящий к файлу объектного кода данной программы. Значением константы *свойвк* является постоянный указатель, приводящий к корневому справочнику внешнего контекста данной программы.

2) *ксн, нпо* — идентификаторы констант, описанных в стандартном контексте (т. е. глобально для всех программ). Значением константы *ксн* является постоянный указатель, приводящий к корневому справочнику контекста съемных носителей (см. п. 5.1). Именование объектов контекста съемных носителей описано в п. 5.4.

Значением константы *нпо* является постоянный указатель, приводящий к непомеченному внешнему объекту. Стратегия постановки нужного непомеченного объекта на устройство определяется оператором.

3) *пустовнеш* — идентификатор константы, описанной в стандартном контексте. Значением этой константы является постоянный указатель, содержащий пустую ссылку.

* **4.1. Форма ссылки на внешний объект.** Ссылка, содержащаяся в указателе внешнего объекта и в элементе справочника, может иметь одну из двух форм:

1. Физическая ссылка, представляющая собой физический адрес объекта по внешнему полю.

2. План поиска, представляющий собой пару, состоящую из ссылки на справочник, с которого начинается поиск, и совокупности параметров поиска. Последняя может представлять собой последовательность алфавитно-цифровых слогов для поиска в контексте, целое число, являющееся номером элемента справочника внешних связей файла и пр. Поиск выполняется не в момент создания плана поиска, а при открытии указуемого объекта, при создании в нем новых объектов (если поиск приводит к контейнеру) и пр. В связи с этим:

— план поиска может, в частности, приводить к такому объекту или элементу справочника, который еще не существует;

— поиск по одному и тому же плану может в разное время приводить к различным объектам.

4.2. Операции над указателем. В описании операций используются следующие обозначения для параметров: *ув* — подвижный указатель; *увпф* — подвижный или постоянный указатель; *во* — объект связи.

Если указатель *ув* установлен на некоторый объект X или элемент справочника X, то это означает, что X прикреплен к *ув*. В результате установки *ув* на другой объект Z происходит отделение X от *ув* и прикрепление Z к *ув*.

1) *обкт* (*ув*, *во*). Пусть *во* — указатель (постоянный или подвижный), установленный на элемент справочника. В этом случае операция устанавливает *ув* на тот объект или элемент справочника, ссылка на который находится в исходном элементе справочника. Если при этом ссылка имеет форму плана поиска, то операция осуществляет поиск. В остальных случаях *ув* устанавливается на внешний объект, непосредственно связанный с *во* (без прохода по промежуточным звеньям). Если *во* является постоянным указателем, содержащим план поиска, то операция осуществляет поиск.

2) *обкт* (*ув*). В этой операции указатель *ув* должен быть установлен на некоторый внешний объект или элемент справочника. При этом условии *обкт* (*ув*) эквивалентно *обкт* (*ув*, *ув*), т. е. если *ув* первоначально установлен на элемент справочника, то в результате операции он устанавливается на объект или элемент справочника, ссылка на который содержится в исходном элементе справочника. В остальных случаях операция дает тождественный результат.

3) *копирув* (*ув*, *во*). Эта операция отличается от *обкт* тем, что она не проходит элемент справочника: *ув* всегда устанавливается на внешний объект или элемент справочника, непосредственно связанный с *во*.

4) *псевдовнеш* (*ув*, *ф*). В этой операции параметр *ф* — это некоторый файл X, содержащий псевдообъект Z или псевдообъект X, содержащий другой псевдообъект Z. В результате операции указатель *ув* устанавливается на псевдообъект Z. В частности, если X является файлом объектного кода, то *ув* устанавливается на содержащуюся в нем программу.

5) *псевдовнеш* (*ув*). В этой операции параметр *ув* должен быть указателем, установленным на файл (или псевдообъект). При этом условии *псевдовнеш* (*ув*) эквивалентно *псевдовнеш* (*ув*, *ув*). *

5. Работа с архивом

В этом параграфе излагаются общие сведения, касающиеся средств работы с архивом, описываются операции над справочниками и конструкция ⟨внешнее имя⟩.

5.1. Общие сведения. Различаются справочники двух разновидностей:

1. Архивный справочник. Это справочник на мб- или мд-носителе, создаваемый с помощью ⟨генератора внешнего объекта⟩. С элементом справочника связано алфавитно-цифровое обозначение или имя. Элементы справочника упорядочены по именам в алфавитном порядке.

* 2. Справочник внешних связей файла (СВС). Такой справочник находится в заголовке файла и создается вместе с файлом. Элементы справочника пронумерованы от 0 до $n-1$, где n — значение атрибута *длинсвс* файла. Объект, обеспечивающий доступ к файлу, одновременно обеспечивает доступ к СВС этого файла. *

Базовый справочник контейнера. В момент создания мд-контейнера в нем автоматически создается архивный базовый справочник данного контейнера. Объект, обеспечивающий доступ к контейнеру, одновременно обеспечивает доступ к базовому справочнику. Используя операции над справочником, можно создавать в базовом справочнике элементы, записывать в них ссылки на объекты и пр.

Некоторые из описанных ниже операций определены над мл-контейнером. В этих случаях мл-контейнер рассматривается как архивный справочник, содержащий элементы, имена которых совпадают с именами файлов этого контейнера. К таким операциям относятся операции поиска по справочнику (*элспр*) и операции позиционирования справочника.

Простые и косвенные ссылки. Различаются две разновидности ссылок из элемента справочника: простые и косвенные. Разновидность ссылки не играет роли при «поиске по справочнику», но играет роль при «поиске по контексту».

Поиск по справочнику. Поиск элемента с именем *s* в архивном справочнике заключается в том, что имена элементов перебираются в алфавитном порядке, с целью найти имя, совпадающее со строкой *s*.

Поиск по контексту. Поиск элемента с именем *s* во внешнем контексте, корневым справочником которого является архивный справочник СПР, происходит следующим образом. Сначала в алфавитном порядке перебираются имена элементов СПР с целью найти элемент с именем, совпадающим со строкой *s*. Если таковой не найден, но в справочнике есть элементы, содержащие косвенные ссылки, то рассматривается первый из них (в алфавитном порядке). Этот элемент должен в свою очередь ссылаться на некоторый архивный справочник СПР1. Алгоритм поиска повторяется в контексте с корневым справочником СПР1. Если в этом контексте имя не найдено, то рассматривается следующий элемент исходного справочника, содержащий косвенную ссылку, и т. д. Глубина рекурсии в алгоритме поиска не должна превышать 10. В противном случае возникает ошибка «нет в архиве».

* *Форма ссылки.* Ссылка из элемента справочника (простая или косвенная) может быть представлена как в форме физической ссылки, так и в форме плана поиска (см. п. 4.1). В случае физической ссылки имеют место следующие ограничения. Физической ссылкой могут быть представлены только:

— взаимные ссылки между мб-объектами;

— взаимные ссылки между мд-объектами, содержащимися в одном и том же контейнере (каталогизированном или некаталогизированном);

— взаимные ссылки между мб-объектом и мд-объектом, содержащимся в каталогизированном (см. ниже) мд-контейнере;
— взаимные ссылки между мд-объектами, содержащимися в различных каталогизированных мд-контейнерах. *

Справочник пользователя. Для каждого пользователя при его регистрации в системе создается корневой справочник внешнего контекста данного пользователя. Пользуясь операциями со справочниками, пользователь может создавать элементы в этом справочнике, заносить в них ссылки на вновь созданные внешние объекты и т. д., т. е. использовать этот справочник в качестве базы для создания личного архива.

Первоначально корневой справочник состоит из одного элемента, имеющего обозначение "*глобарх*". Этот элемент содержит косвенную ссылку на глобальный внешний контекст, доступный всем пользователям.

Обращение к корневому справочнику происходит следующим образом. Если программа является программой пакетного задания, то внешним контекстом программы является контекст пользователя, сформировавшего задание (пользователь идентифицируется «картой пользователя», см. § 15). Следовательно, используя в задании идентификатор константы *свойвк*, можно получить указатель, приводящий к корневому справочнику внешнего контекста пользователя.

* *Каталогизация.* Значение атрибута *катлг* предписывает каталогизировать внешний объект (мд-, мл-контейнер). При каталогизации метка объекта (см. атрибуты *имяконт* и *имяфайла*) и другие сведения о нем помещаются в каталог системы. *

Съемные объекты. Именование объектов на съемных носителях (мд- и мл-контейнеры, мл-, чпк- и чпл-файлы) трактуется следующим образом. Предполагается, что существует глобальный известный контекст съемных носителей. У этого контекста есть гипотетический корневой справочник. Значением константы *ксн*, описанной глобально для всех программ, является постоянный указатель, установленный на этот справочник. Поиск элемента с именем *s* состоит в том, что оператору выдается запрос на установку съемного носителя, содержащего объект с меткой *s* (см. также пп. 2.1 и 5.4). Такой способ обращения предназначен как для каталогизированных, так и для некаталогизированных (приносимых) объектов.

5.2. Операции над справочником. В описании операций используются следующие обозначения для параметров: *ув* — подвижный указатель, *во* — объект связи, *спр* — архивный справочник, контейнер или файл. В последних двух случаях опера-

ция выполняется соответственно над базовым справочником контейнера или над СВС файла; *спрарх* — архивный справочник или контейнер; *им* — параметр, который может быть (а) набором, возможно неполным, или указателем литерной строки (в этом случае параметр задает обозначение элемента архивного справочника) или (б) целым числом (в этом случае параметр задает номер элемента СВС); *уэс* — указатель (подвижный или постоянный), приводящий к элементу справочника.

* Если элемент справочника содержит ссылку на объект X в форме физической ссылки, то это означает, что X прикреплен к данному справочнику. В результате записи в данный элемент справочника новой ссылки происходит открепление X. Если эта новая ссылка представлена в форме физической ссылки на объект Z, то Z прикрепляется к справочнику.

1) *элспр* (*ув*, *спр*, *им*), *элвк* (*ув*, *спрарх*, *им*). Операция *элспр* осуществляет поиск элемента, с обозначением *им*, в справочнике *спр*. Операция *элвк* осуществляет поиск элемента, с обозначением *им*, в контексте с корневым справочником *спрарх*.

Результатом операции является указатель *ув*, установленный на найденный элемент справочника. Если же элемент не найден, то возникает ошибка «нет в архиве».

2) *элспр* (*ув*, *им*), *элвк* (*ув*, *им*). Эти операции отличаются от приведенных выше тем, что указатель *ув* первоначально должен быть установлен на справочник, в котором производится поиск. Тогда:

элспр (*ув*, *им*) эквивалентно *элспр* (*ув*, *ув*, *им*)

элвк (*ув*, *им*) эквивалентно *элвк* (*ув*, *ув*, *им*) *

3) *создэлспр* (*спрарх*, *им*). Операция создает в архивном справочнике *спрарх* новый элемент. Обозначение элемента задается параметром *им*. В созданный элемент справочника заносится пустая ссылка.

Результатом операции является значение параметра *спрарх*.

4) *создэлспр* (*спрарх*, *им*, *во*), *создэлвк* (*спрарх*, *им*, *во*). Данные операции отличаются от приведенных выше тем, что созданный элемент инициализуется, т. е. в него заносится ссылка на внешний объект или элемент справочника, непосредственно связанный с объектом *во*.

В элемент справочника нельзя заносить ссылку на константный файл или константный справочник. Операция *создэлспр* заносит простую ссылку, а *создэлвк* — косвенную.

Результатом является значение параметра *спр*.

* В зависимости от разновидности параметра *во* ссылка имеет следующую форму:

— постоянный указатель. Ссылка представлена в форме плана поиска из этого указателя;

— оперативный объект, связанный физической ссылкой с некоторым внешним объектом или элементом справочника. Ссылка представляет собой копию физической ссылки из объекта связи. *

5) *запспр* (*уэс*, *во*), *запвк* (*уэс*, *во*). В элемент, на который установлен *уэс*, помещается ссылка на внешний объект или элемент справочника, непосредственно связанный с *во*. Операция *запспр* заносит простую ссылку, а операция *запвк* — косвенную.

Результатом является значение параметра *уэс*.

* Форма ссылки зависит от параметра *во* и определяется так же, как описано выше.

6) *элспр* (*пув*, *спр*, *мсим*). Операция создает программный указатель внешнего объекта. Параметры имеют следующие значения: *пув* — указатель с-вектора из двух элементов формата ф64; *мсим* — указатель литерной строки или набор. Эта строка (или набор) содержит последовательность алфавитно-цифровых слогов, подчиняющуюся синтаксису ⟨многослогового имени⟩ (см. п. 5.4).

Операция записывает в первый элемент вектора значение первого параметра, а второй — значение второго параметра. Эта пара образует программный указатель, приводящий к некоторому внешнему объекту, находящемуся в контексте с корневым справочником *спр*. В качестве результата выдается указатель объекта связи, описывающий вектор *пув* как постоянный указатель внешнего объекта.

5.3. Операция позиционирования справочника. В операциях позиционирования справочника используется подвижный указатель, установленный в некоторую позицию справочника (УПС). УПС, так же как и указатель, установленный на элемент справочника, идентифицирует некоторый элемент. Различие их состоит в том, что в первом содержится признак доступа ко всему справочнику, и такой указатель можно с помощью приведенных ниже операций перемещать с одного элемента справочника на другой элемент того же справочника. Во втором случае переместить указатель с одного элемента на другой можно только с помощью операции *элспр*, т. е. в программе должен быть доступен не только какой-либо элемент справочника, но и весь справочник.

При выполнении операций *начспр* или *конспр* (установить на начало справочника, установить на конец справочника) в указатель заносится признак доступа. Операции *следэлспр* и *предэлспр* воспринимают УПС как указатель исходной позиции и устанавливают его в соседнюю позицию. Операция *текэлспр* аннулирует признак доступа ко всему справочнику. В операциях, отличных от операций позиционирования, *упс* эквивалентен указателю, установленному на элемент справочника.

В описании операций используются следующие обозначения: *ув*, *спр* — см. операции над справочником; *упс* — указатель позиции в справочнике,

1) *начспр*(*ув*, *спр*), *конспр*(*ув*, *спр*). Операция *начспр* устанавливает указатель *ув* на начало справочника, а операция *конспр* — на конец справочника.

Результатом операции является указатель *ув*, в котором установлен признак доступа.

2) *начспр*(*ув*), *конспр*(*ув*). Указатель *ув* должен быть установлен на справочник. Тогда:

начспр(*ув*) эквивалентно *начспр*(*ув*, *ув*)

конспр(*ув*) эквивалентно *конспр*(*ув*, *ув*)

3) *следэлспр* (*ув*, *упс*). В процессе выполнения операции различаются следующие случаи:

— *упс* установлен на начало справочника. В этом случае *ув* устанавливается на первый элемент справочника;

— *упс* установлен на некоторый элемент справочника. В этом случае *ув* устанавливается на следующий элемент;

— *упс* установлен на последний элемент или на конец справочника. В этом случае возникает ошибка «нет в архиве». *ув* устанавливается на конец справочника.

Результатом является значение параметра *ув*.

4) *предэлспр*(*ув*, *упс*). Семантика этой операции аналогична операции *следэлспр* с учетом замены направления перемещения на противоположное.

5) *текэлспр*(*ув*, *упс*). *ув* устанавливается на тот же элемент, что и *упс*, но в *ув* не ставится признак доступа ко всему справочнику. *упс* в этой операции не может быть установлен на начало справочника или на конец справочника.

Результатом является значение параметра *ув*.

6) *начспр*(*ув*), *конспр*(*ув*), *следэлспр*(*упс*), *предэлспр*(*упс*), *текэлспр*(*упс*). Указатель *ув* первоначально должен быть установлен на справочник. Тогда:

начспр(*ув*) эквивалентно *начспр*(*ув*, *ув*)

конспр(*ув*) эквивалентно *конспр*(*ув*, *ув*)

следэлспр(*упс*) эквивалентно *следэлспр*(*упс*, *упс*)

предэлспр (*упс*) эквивалентно *предэлспр* (*упс*, *упс*)

текэлспр(*упс*) эквивалентно *текэлспр*(*упс*, *упс*) *

5.4. Внешнее имя. ⟨Внешнее имя⟩ используется для обозначения постоянного указателя внешнего объекта.

⟨внешнее имя⟩ ::= {⟨идентификатор⟩} ⟨план поиска⟩
⟨план поиска⟩ ::= ⟨многослоговое имя⟩ | // [⟨выражение⟩]
⟨многослоговое имя⟩ ::= ⟨слог⟩ ...
⟨слог⟩ ::= // ⟨буква или цифра⟩ ... |.//
⟨стандартный слог⟩
⟨стандартный слог⟩ ::= . *обкт* | . текст | . *код* | . *псевдо*

⟨Внешнее имя⟩ в виде ⟨многослогового имени⟩, не содержащего ⟨стандартных слогов⟩, обозначает указатель, установленный на элемент некоторого справочника. Последний находится во внешнем контексте программы.

Упомянутый элемент справочника можно найти следующим образом. Пусть ⟨внешнее имя⟩ имеет вид

$$//s_1//s_2// \ldots //s_n$$

где s_k — k-й слог, являющийся алфавитно-цифровым именем. Сначала во внешнем контексте ищется (см. п. 5.1) такой элемент справочника, обозначение которого совпадает с первым слогом (при неудовлетворительном результате поиска возникает ошибка «нет в архиве»). Если во ⟨внешнем имене⟩ есть второй слог, то найденный элемент справочника должен в свою очередь ссылаться на справочник (возможно, через посредство промежуточных звеньев). Этот справочник рассматривается как корневой, и процесс поиска повторяется уже со вторым слогом. Поиск по третьему и последующим слогам выполняется аналогично. В результате в некотором справочнике будет найден элемент, имя которого совпадает с последним слогом. Это и есть искомый элемент. Смысл ⟨стандартных слогов⟩ описан ниже.

Заметим, что описанный здесь алгоритм выполняется при непосредственном взаимодействии с объектом, напримерр, в момент открытия. При выполнении конструкции ⟨внешнее имя⟩ поиск не происходит, а лишь создается указатель, содержащий план поиска.

* Ссылка на внешний контекст программы находится в справочнике внешних связей файла объектного кода программы. Эту ссылку заносит транслятор после генерации файла или пользователь. Номер элемента, в котором должна содержаться ссылка, определяется во время трансляции и заносится в атрибут *внешконт*. *

Съемный объект. Указатель, приводящий к объекту контекста съемных носителей, можно задать с помощью имени из метки объекта (см. атрибуты *имяфайла*, *имяконт*). При этом используется ⟨внешнее имя⟩, начинающееся с ⟨идентификатора⟩ константы или переменной, значением которой является указатель, установленный на корневой справочник контекста съемных носителей (см. п. 5.1). В простейшем случае — это константа *ксн*. Например, к контейнеру с некоторым именем "*моймд*" можно обращаться с помощью ⟨внешнего имени⟩ *ксн//моймд*.

* *Объект произвольного внешнего контекста*. ⟨Внешнее имя⟩ может начинаться с ⟨идентификатора⟩ константы или переменной (отличной от формального параметра), значением которой является некоторый справочник X. В этом случае ⟨внешнее имя⟩ обозначает указатель, установленный на элемент справочника, находящегося во внешнем контексте, корне-

вой справочник которого есть X. Поиск нужного элемента в этом контексте производится, как описано выше.

Элемент СВС. ⟨Внешнее имя⟩ с ⟨планом поиска⟩, имеющим вид //[⟨выражение⟩], обозначает указатель, установленный на элемент СВС некоторого файла. ⟨Идентификатор⟩ в этом случае задает файл, а значение ⟨выражения⟩ — номер элемента в СВС этого файла. При опущенном ⟨идентификаторе⟩ рассматривается файл объектного кода данной программы.

⟨*Стандартные слоги*⟩ интерпретируются следующим образом:

—.*обкт* равносилен применению операции *обкт* к предшествующей ему части ⟨внешнего имени⟩, т. е. данный слог приводит к тому объекту или элементу справочника, на который установится подвижный указатель, если к нему применить операцию: *обкт* (*ув*, *s*), где *ув* — некоторый подвижный указатель, *s* — часть ⟨внешнего имени⟩, предшествующая рассматриваемому слогу;

—.*текст* (.*код*, .*псевдо*). Часть ⟨внешнего имени⟩, предшествующая такому слогу, должна приводить к файлу объектного кода (файлу текста, файлу, содержащему псевдообъект). В этом случае слог приводит к элементу СВС файла, содержащему ссылку на текст программы (к элементу СВС файла, содержащему ссылку на код программы, к псевдообъекту). Иными словами, слог приводит к тому элементу справочника или к тому объекту, на который установится подвижный указатель, если к нему применить операцию:

элспр (*ув*, *s*, *читатр* (*s*, *текстпрогр*))

(*элспр* (*ув*, *s*, *читатр* (*s*, *кодпрогр*)), *псевдовнеш* (*ув*, *s*))

Ограничения. Один ⟨слог⟩ не должен содержать больше 17 букв и/или цифр. ⟨Многослоговое имя⟩ не должно содержать пробелов. *

6. Средства работы с мл-контейнером

6.1. Текущая позиция. Текущая позиция в мл-контейнере — это положение головки записи/считывания по отношению к началу контейнера. Том (файл, зона), на который установлена головка, называется текущим томом (текущим файлом, текущей зоной). Текущая позиция характеризуется значениями позиционных атрибутов контейнера *томконт, файлконт, имяфайла, нмртома, позтома* и позиционным атрибутом файла *блокфайла*. При модификации позиционного атрибута текущая позиция перемещается. Названные атрибуты контейнера имеют следующий смысл:

томконт — порядковый номер текущего тома относительно начала контейнера (математический номер). При модификации этого атрибута текущая позиция перемещается на том с номером, равным новому значению атрибута. Если данный том уже смонтирован на устройство, то положение головки записи/считывания сохраняется. В противном случае текущая пози-

ция перемещается на начало тома. Кроме целочисленного значения, этот атрибут может принимать следующие значения типа набор: "*начк*" — начало контейнера, т. е. начало первого тома контейнера, и "*конк*" — конец контейнера, т. е. конец последнего тома контейнера.

файлконт — порядковый номер текущего файла относительно начала контейнера. При модификации этого атрибута текущая позиция перемещается на файл, номер которого равен новому значению атрибута. Если том с нужным файлом уже смонтирован на устройство, то в зависимости от исходного местоположения головки относительно файла текущая позиция может установиться как на начало, так и на конец файла. В противном случае файл устанавливается на начало.

имяфайла — имя из метки текущего файла. Если атрибуту присвоить литерный набор или литерную строку, то текущая позиция перемещается на файл с заданным (этим набором или строкой) именем. Текущая позиция внутри файла определяется, как и в случае модификации атрибута *файлконт*.

нмртома — регистрационный номер текущего тома. Модификация атрибута имеет смысл, аналогичный модификации атрибута *имяфайла* с учетом того, что текущая позиция перемещается на том с заданным регистрационным номером.

Конструкция ⟨модификация атрибутов⟩ допускает одновременное изменение нескольких позиционных атрибутов. В частности, для избежания неопределенности в установке текущей позиции можно одновременно с атрибутами *томконт*, *файлконт*, *имяфайла* или *нмртома* задавать атрибут *поztома*. Например, модификация:

запатр (*млк1*, *файлконт* : 10, *поztома*: "*начф*") ...

устанавливает текущую позицию контейнера *млк1* на начало файла с номером 10.

Атрибуту *поztома* можно придавать следующие значения типа набор: "*начт*" — начало тома, "*конт*" — конец тома, "*начф*" — начало файла, "*конф*" — конец файла.

6.2. Назначение физических томов. Если в генераторе контейнера указан атрибут *имяконт*, то создается сохраняемый контейнер, т. е. такой контейнер, который не ликвидируется при закрытии. Если в генераторе не задан атрибут *нмртома*, то в качестве первого тома назначается любой из свободных. В противном случае назначается том с указанным регистрационным номером. В метку тома записываются имя контейнера, имя пользователя и порядковый номер 1 (атрибуты *имяконт*,

имяпол и *томконт*). Текущая позиция устанавливается на начало первого тома. Например, в результате выполнения оператора.

*млк*1 : = *генконтейнер* (*млконт*, *имяконт*; "*моймлк*", *нмртома* : "*мл*100")

создается мл-контейнер с именем "*моймлк*". В качестве первого тома назначается лента с регистрационным номером мл100.

Вновь созданный контейнер прикрепляется к контексту съемных носителей. Это значит, что при закрытии он не ликвидируется. В дальнейшем к нему можно обращаться по ⟨внешнему имени⟩ *ксн*//*s*, где *s* — имя метки. Например, *ксн*//*моймлк* является ⟨внешним именем⟩ контейнера «*моймлк*». Если значением атрибута *имяконт* является пустая строка, то контейнер считается непомеченным, и к нему можно обращаться через стандартный указатель *нпо*.

Если в генераторе контейнера атрибут *имяконт* не указан, то создается временный контейнер, т. е. такой контейнер, который ликвидируется при закрытии. Если в генераторе не задан атрибут *нмртома*, то в качестве первого тома назначается любой из свободных. В противном случае назначается том с указанным регистрационным номером. В метку тома записываются имя пользователя и порядковый номер 1. Кроме того, для технических нужд системы формируется распознаваемое системой имя контейнера, которое также записывается в метку. Текущая позиция устанавливается в начало первого тома.

Стратегия назначения последующих томов контейнера (сохраняемого или временного) зависит от того, в каком режиме затребуется следующий том.

Если том назначается в режиме записи (запись нового блока, не умещающегося на текущем томе, или создание нового файла, метка которого не умещается на текущем томе), то возможны два варианта:

1. Можно заранее назначить математическим томам физические, указав для каждого математического тома регистрационный номер соответствующего физического. Для этого нужно воспользоваться модификацией атрибутов тома, например:

запатр(*млк*1, 2, *нмртома*: "*т*112").

запатр(*млк*1, 3, *нмртома*: "*т*113").

Здесь 2-му и 3-му тому контейнера ставятся в соответствие физические тома с регистрационными номерами *т*112 и *т*113.

2. Если соответствие не указано, то в качестве нового тома назначается любой из свободных.

В метку нового тома записываются имя контейнера (для временных — техническое имя), имя пользователя и порядковый номер тома.

Если том запрашивается в режиме чтения (чтение блока и установка текущей позиции на блок, расположенный вне текущего тома, или установка текущей позиции на файл, расположенный вне текущего тома), то выбирается ранее созданный следующий том данного контейнера.

Создание файла. Новый файл всегда создается в текущей позиции контейнера. В связи с этим в общем случае для создания файла необходимо предварительно установить нужную позицию, а затем уже произвести генерацию объекта. В качестве основы генератора подается контейнер. В зависимости от исходной позиции различаются 4 случая:

1. Начало тома — создается первый файл тома.

2. Конец тома — создается файл, непосредственно следующий за последним из уже существующих файлов тома.

3. Начало файла — старый файл затирается, и на его месте создается новый.

4. Конец файла — создается файл, непосредственно следующий за текущим.

Вновь созданный файл и том, на котором он создан, считаются последними в контейнере. Текущая позиция устанавливается на начало нового файла.

6.3. Доступ к файлу. Для открытия существующего файла в контейнере необходимо предварительно установить на него указатель внешнего объекта. Это можно сделать несколькими способами:

1. Задать постоянный указатель с помощью внешнего имени *ксн//s1//s2*, где *s1* — имя из метки контейнера, а *s2* — имя из метки файла, или *x//s2*, где *x* — контейнер.

* 2. Установить подвижный указатель с помощью операции поиска элемента справочника *элспр*. В операцию подается контейнер (в качестве справочника, см. п. 5.1) и имя искомого файла. В этом случае текущая позиция устанавливается на нужный файл уже при выполнении операции поиска *). Результирующая позиция определяется, как описано выше для случая модификации атрибута *имяфайла*.

3. Установить подвижный указатель с помощью операции позиционирования справочника. Как и в предыдущем случае, аргументом операции является контейнер. В процессе выполнения операции текущая позиция устанавливается на нужный

---

*) Для мл-контейнера не могут одновременно существовать два подвижных указателя, установленных в разные позиции.

файл. Результирующая позиция определяется, как описано выше для случая модификации атрибута *томконт*.

4. Модифицируя позиционные атрибуты контейнера, установить текущую позицию на начало или на конец нужного файла. Затем, используя операцию копирования указателя *копирув*, установить в эту же позицию указатель внешнего объекта, например:

копирув(*ув*, *запатр*(*мл1*, *файлконт*: 10)),

где *ув* и *млк1* — соответственно указатель и контейнер. *

Указатель, полученный одним из перечисленных способов, (постоянный или подвижный) установлен на закрытый млфайл. Чтобы открыть файл, указатель надо подать в операцию открытия, например:

млф := *файл* (*ув*)

Если здесь используется постоянный указатель, то сначала производится поиск нужного файла, а затем его открытие. Текущая позиция первоначально устанавливается так, как описано выше для случая модификации атрибута *имяфайла*.

6.4. Закрытие файла. При перемещении текущей позиции с одного файла на другой первый автоматически закрывается (если он был открыт).

При перемещении текущей позиции с одного тома на другой текущий файл первого закрывается. Том не снимается с устройства, головка остается в том положении, куда она была помещена при закрытии файла. Оператор, однако, может перемотать и снять том, поэтому система не гарантирует фиксацию положения томов контейнера, кроме положения текущего тома.

Перемещение текущей позиции, происходящее при закрытии файла, зависит от атрибута *направление*. Значение **истина** (**ложь**) задает прямое (обратное) направление обработки, и текущая позиция перемещается на конец (на начало) закрываемого файла.

Если файл занимает несколько томов и направление обработки прямое (обратное), то при закрытии файла текущая позиция перемещается на конец (начало) файла на текущем томе.

7. Обмен. Общие сведения

В табл. 1 указаны возможные способы организации обмена. В правой колонке таблицы на примере операции чтения представлен образец конструирования операции для каждого конкретного способа обмена.

Таблица 1. **Способы двоичного обмена.**

обмен	буферизованный	однобуферный	последов.:		*чит (пзп, след (к))*
			пропзв.:	спнхр.:	*чит (пзп, нбл)*
				парал.:	*читпар (пзп, нбл)*
		многобуферный	последов.:		*чит (тбф, буф, след (к))*
			пропзв.:	спнхр.:	*чит (тбф, нбл)*
				парал.:	*читпар (тбф, нбл)*
	непосредственный	ацд, пм, ацпу	последов.:	спнхр.:	*чит (ф, след (к), м)*
				парал.:	*читпар (бвф, след (к), м)*
		мл, пк, пл			
		мб, мд	пропзв.:	спнхр.:	*чит (ф, а, м)*
				парал.:	*читпар (бвф, а, м)*

Сокращения: — послед.— последовательный доступ;
— пропзв.— пропзвольный доступ;
— спнхр.— спнхронный режим обмена;
— парал.— параллельный режим обмена.

7.1. Операции обмена. Конструкция ⟨двоичный обмен⟩ предназначена для осуществления операций ввода информации с файла, вывода информации на файл и позиционирования файла. В отличие от ⟨форматного обмена⟩ эти операции не производят редактирование данных.

Ниже описываются наиболее общие свойства операций и даются ссылки на параграфы, в которых представлена подробная семантика для каждого конкретного способа обмена.

⟨двоичный обмен⟩ ::=
⟨идентификатор операции⟩ (⟨выражение⟩ ,
⟨спецификация адреса⟩ { , ⟨выражение⟩}
{ , *ошобмена* : ⟨выражение⟩})
⟨идентификатор операции⟩ ::=
чит | *читпар* | *зап* | *заппар* | *нов* | *уст* | *устпар*
⟨спецификация адреса⟩ ::=
⟨выражение⟩
| ⟨направление⟩ {(⟨выражение⟩)}
| ⟨выражение⟩ , ⟨направление⟩ {(⟨выражение⟩)}
| *нач* | *кон* | *канал* (⟨выражение⟩)
⟨направление⟩ ::= *след* | *пред*

Способы обмена. Значением первого ⟨выражения⟩ может быть:

— позиционная переменная. В этом случае выполняется одна из операций однобуферного способа буферизованного обмена (см. § 10);

— таблица буферов. В этом случае выполняется одна из операций многобуферного способа буферизованного обмена (см. § 11);

— заголовок открытого файла. Выполняется одна из операций синхронного непосредственного обмена (см. § 8);

— блок ввода/вывода. В этом случае выполняется одна из операций параллельного непосредственного обмена (см. § 8).

Обозначение операции. ⟨Идентификатор⟩ операции задает операцию, выполняемую над объектом:

— *чит, читпар* — операции чтения (синхронного, параллельного);

— *зап, заппар* — операции записи (синхронной, параллельной);

— *нов* — дополнительная операция записи без предварительного считывания блока в буфер;

— *уст, устпар* — операции позиционирования (синхронного, параллельного);

Тип доступа. Существуют операции для последовательного доступа и операции для произвольного доступа. Тип доступа (последовательный, произвольный) определяет ⟨спецификация адреса⟩. Там же указывается участок файла, подлежащий обработке:

— форма ⟨выражение⟩ используется в операциях произвольного доступа. Значение ⟨выражения⟩ задает адрес начала обмена. Такой тип доступа возможен для мб- и мд-файлов (см. пп. 8.1, 10.3, 11.1) и для мл-файлов при буферизованном обмене (см. п. 10.3).

— форма *след* (⟨выражение⟩) используется в операциях последовательного доступа. Значение ⟨выражения⟩ задает смещение текущей позиции. Такой тип доступа возможен для физически последовательных файлов (см. пп. 8.3—9.6) и при буферизованном обмене для файлов всех типов (см. § 10). Если указано ⟨направление⟩ *пред*, то обработка производится в обратном направлении (от конца файла к началу). Если скобки и ⟨выражение⟩ опущены, то подразумевается смещение, равное 1.

— форма из двух компонентов ⟨выражение⟩, *след* (⟨выражение⟩) используется в операциях последовательного доступа при многобуферном способе буферизованного обмена (см. п. 11.2). В этом способе считается, что существует несколько текущих позиций, и с каждой из них связан буфер. Значением

первого ⟨выражения⟩ является один из этих буферов, а значение второго ⟨выражения⟩ задает смещение текущей позиции, соответствующей этому буферу. Если указано направление *пред*, то обработка производится в обратном направлении. Если скобки и второе ⟨выражение⟩ опущены, то подразумевается смещение, равное 1;

— формы *нач*, *кон* задают в операциях позиционирования смещение текущей позиции на начало файла и на конец файла (см. пп. 8.3, 8.4, 10.1, 11.2);

— форма *канал* (⟨выражение⟩) представляет собой специфическое указание позиции, предназначенное только для ацпу-файлов (см. пп. 8.5, 10.1).

Массив обмена. При непосредственном обмене (см. § 8) в операциях чтения/записи необходимо задавать массив обмена. В данном случае допускается только с-вектор. Указатель вектора является значением ⟨выражения⟩, следующего за ⟨спецификацией адреса⟩. В операциях чтения информация считывается из файла в этот вектор, а в операциях записи информация из этого вектора записывается в файл. При буферизованном обмене эта компонента не указывается, так как обмен происходит с буфером.

Результат. Операции, отличные от операций буферизованного чтения (записи), в качестве результата выдают первый параметр. Операции буферизованного чтения/записи выдают указатель, описывающий буферный массив.

Ошибки обмена. Компонента, начинающаяся со слова *ошобмена*, предназначена для того, чтобы указать переопределение стандартных системных реакций на ошибки обмена. Это переопределение имеет силу на время выполнения данной операции обмена. Значением ⟨выражения⟩ должен быть указатель массива процедур реакций на ошибки обмена (см. гл. 5).

7.2. Назначение буферизованного обмена. При буферизованном способе обмена взаимодействие с памятью файла осуществляется с помощью промежуточных буферов. При этом считается, что файл логически состоит из блоков, упорядоченных по номерам. В зависимости от типа внешнего устройства различаются файлы, состоящие из блоков постоянной длины, и файлы, состоящие из блоков переменной длины.

Однобуферный обмен. Существуют два способа буферизованного обмена: однобуферный и многобуферный. Однобуферный способ в основном предназначен для реализации традиционного последовательного доступа. В этом случае с файлом связана позиционная переменная. Позиционная переменная

представляет собой объект связи с файлом и создается с помощью генератора, например:

*поз*1 := *позп* (*мдф*1),

здесь создается позиционная переменная для буферизованного обмена с файлом *мдф*1;

*поз*2 := *позп* (//*файл*2),

здесь создается позиционная переменная для буферизованного обмена с файлом, доступным по внешнему имени //*файл*2.

Считается, что при буферизованном обмене с файлом существует текущая позиция, характеризующая положение головки записи/считывания по отношению к началу файла. Головка установлена на текущий обрабатываемый блок. Позиционная переменная содержит информацию о текущей позиции, а именно: значение ее атрибута *блокфайла* — это номер текущего блока.

С позиционной переменной связан текущий буфер. Этот буфер является отображением текущего блока в оперативной памяти. В результате выполнения операции чтения или записи текущая позиция сдвигается на следующий (или предыдущий — в зависимости от направления обработки) блок. Этот блок становится текущим, и выдается указатель соответственно для считывания из буфера содержимого текущего блока или для записи в буфер текущего блока, например:

*буф*1 := *чит* (*поз*1, *след*)
*буф*2 := *зап* (*поз*2, *след*)

Используя полученный указатель, можно записывать или считывать значения элементов обычным образом:

*x*1 := *буф*1 [3] % считывание из буфера
*a*1 := *буф*1 длиной *к* % считывание из буфера
*буф*2 [*i*] := *x*2 % запись в буфер
*буф*2 <:= *a*2 % запись в буфер

Операция чтения всегда производит считывание соответствующего блока в буфер. В отличие от этого, операция записи производит считывание блока только в том случае, если он находится в пределах заполненной части файла. Если же в заполненной части файла необходимо полностью заменить некоторый блок, т. е. нет необходимости считывать его старое содержимое, то для этого нужно использовать другую операцию записи:

дз := *нов* (*поз*2, *след*)

В этом случае выдается указатель для записи, но блок не считывается в буфер.

Если в буфер производилась запись, то по окончании работы с этим буфером система осуществляет автоматический сброс содержимого буфера в файл.

После того как произошел сдвиг текущей позиции с одного блока на другой, доступ к буферу первого становится невозможен. Таким образом, в данном способе в каждый момент можно иметь доступ только к одному, а именно текущему блоку файла.

* *Многобуферный обмен.* Многобуферный способ буферизованного обмена осуществляется через объект «таблица буферов». Эта таблица создается с помощью генератора, например:

*многобуфмдф*1 := *тбуф* (*мдф*1)
*многобуфмдф*2 := *тбуф* (//*файл*2)

В многобуферном способе не существует специальной позиционной переменной. Позиция указывается с помощью номера блока.

Как и в однобуферном способе, в результате обращения к некоторому блоку выдается указатель буфера этого блока, но при этом не ликвидируется возможность доступа к буферам, которые обрабатывались перед этим. Например, последовательность двух операторов:

*дбл*1 := *чит*(*многобуфмдф*1, *нбл*1)
*дбл*2 := *чит*(*многобуфмдф*1, *нбл*2)

дает возможность одновременно работать с двумя буферами, а именно: с помощью *дбл*1 — с буфером, соответствующим блоку *нбл*1, и с помощью *дбл*2 — с буфером, соответствующим блоку *нбл*2. Каждый буфер остается заблокированным до тех пор, пока программа не осуществит его явную разблокировку, например:

открепбуф (*многобуфмдф*1, *дбл*1)

Функции системы при буферизованном обмене. Система обеспечивает создание, уничтожение и переиспользование буферов, а также предварительное их заполнение блоками файла (параллельно с выполнением программы) при последовательном доступе. *

7.3. Назначение непосредственного обмена. Процедуры непосредственного обмена дают возможность реализовать пересылку данных между памятью программы и файлом без какого-либо промежуточного хранения. Непосредственный обмен можно осуществлять как синхронно с работой программы, так и параллельно.

* *Режим обмена.* Семантическое отличие синхронного непосредственного обмена от параллельного заключается в следующем. В первом случае к моменту завершения выполнения опе-

рации обмена соответствующая операция над внешним носителем уже выполнена. Во втором случае операция только инициирует начало обмена. В случае строго последовательных внешних устройств (мл, пк, ацпу, пл) при параллельном обмене сохраняется порядок выполнения реальных операций, но по времени они в общем случае запаздывают по отношению к процессу выполнения программы. При необходимости это запаздывание можно ликвидировать, используя в программе операции синхронизации над семафором завершения обмена.

Блок ввода/вывода. Одной из основных функций системы при реализации некоторой конкретной пересылки данных между оперативной памятью и файлом является создание так называемого блока ввода/вывода (БВВ). БВВ, как и позиционная переменная, является объектом связи с файлом. Назначение БВВ состоит в том, что он связывает файл и тот массив в оперативной памяти, с которым необходимо произвести обмен. Кроме того, в БВВ хранится информация об операции, о местоположении обрабатываемого участка файла, о реакциях на возможные ошибки обмена и пр.

С точки зрения параметров операций различие между синхронным и параллельным непосредственным обменом состоит в следующем. В первом случае система неявно формирует БВВ на время данного обмена, и первым параметром процедуры обмена является заголовок открытого файла, например:

*диск*1 := *файл*(//*файл*2)

Здесь открывается файл, доступный по внешнему имени //*файл*2, и значением переменной *диск*1 становится заголовок открытого файла;

чит(*диск*1, *адр*1, *мас*1),

здесь информация из файла, начиная с адреса *адр*1, читается в массив *мас*1.

Во втором случае сама программа должна явно создавать БВВ с помощью генератора, например:

*бввдиск*2 := *бвв*(//*файл*2)

и использовать его в качестве первого параметра процедуры обмена:

читпар(*бввдиск*2, *адр*2, *мас*2)

Одним из атрибутов БВВ является семафор завершения обмена. Запросив этот атрибут, можно реализовать синхронизацию с процессом обмена:

ждать(*читатр*(*бввдиск*2, *смфобмена*)) *

7.4. Ошибки обмена. В процессе обмена могут возникнуть особые обстоятельства, объединяемые под общим названием «ошибки обмена». Часть из них имеет аварийный характер и требует либо аварийного прекращения задачи, либо выполнения каких-либо аварийных действий по перезапуску обмена, отмены операции обмена и пр. Другая часть этих особых обстоя-

тельств не имеет аварийного характера, и в этих случаях система не предпринимает действий, приводящих к прекращению задачи.

Стандартная реакция системы на каждую из ошибок может быть заменена на реакцию, определенную пользователем. В гл. 5 описана номенклатура ошибок, стандартные реакции на ошибки, а также средства переопределения стандартной реакции. Обстоятельства, при которых возникают ошибки, отмечены в описании семантики операций обмена.

7.5. Обозначения, используемые при описании операций. В описании семантики операций используются следующие обозначения для параметров:

ф (*пзп*, *тбф*, *бвф*) — заголовок открытого файла (позиционная переменная, таблица буферов, блок ввода/вывода);

буф — указатель, частично или полностью описывающий буферный массив. Этот указатель идентифицирует буфер, над которым выполняется операция;

а — номер сектора мб- или мд-файла в операциях непосредственного обмена;

нбл — номер блока в операциях буферизованного обмена;

м — массив в операциях непосредственного обмена;

к — величина смещения текущей позиции.

Кроме того, используются следующие обозначения:

блкф — номер первого незаполненного блока файла;

пзп'блокфайла — атрибут *блокфайла* из позиционной переменной пзп. Аналогичным образом обозначаются другие атрибуты позиционной переменной, файла, БВВ и буферного массива.

8. Непосредственный обмен

Наличие специфических особенностей у внешних устройств различного типа приводит к тому, что трактовка операций непосредственного обмена зависит от типа устройства, на котором расположен файл. В связи с этим описание непосредственного обмена ориентировано на уровень внешних устройств и разбито на пункты, соответствующие типам устройств.

8.1. Мб- и мд-файлы. Ввиду того, что принципы организации и средства обмена с мб-файлами и мд-файлами схожи между собой, в последующем описании, если нет необходимости их различать, будет употребляться термин «мбмд-файл» и «мбмд-носитель».

Физическая адресация к мбмд-носителю осуществляется с точностью до сектора. Физический объем сектора мб-носителя —

32 72-разрядных слова, а сектора мд-носителя — 32 64-разрядных слова. Сектор мд-носителя, в зависимости от значения атрибута *типданных*, может логически состоять из 32 или 28 элементов.

Для реализации распределения физической памяти на мбмд-носителе используется метод листового распределения памяти, т. е. математическим листам файла ставятся в соответствие физические листы. Физический лист представляет собой совокупность нескольких смежных секторов. Все математические листы некоторого конкретного файла упорядочены по номерам и имеют одинаковую длину, равную длине физического листа этого файла. Длина физического листа определяется значением атрибута *длинлиста* для данного файла (для разных файлов на данном носителе это значение может быть различно). Значение этого атрибута задает количество секторов в листе. Максимально возможное количество листов файла определяется значением атрибута *максдлинфайла*.

* В общем случае физические листы некоторого файла располагаются на носителе несмежно. Отведение физического листа, соответствующего некоторому математическому листу, осуществляется динамически, в момент первой записи в этот лист. Однако с помощью атрибутов *порядоклистов*, *листвцилиндр*, *класслиста* и *файлвпакет* можно управлять расположением физических листов на носителе, а с помощью атрибута *распределен* — отведением и освобождением физических листов.

Операция *перестлист* ($\phi1$, $к1$, $\phi2$, $к2$) переставляет местами лист с номером $к1$ из файла $\phi1$ и лист с номером $к2$ из файла $\phi2$. При этом физической переписи листов не происходит, а осуществляется перестановка базовых адресов листов в таблицах листов указанных файлов. На параметры накладываются следующие ограничения: значения атрибута *длинлиста* у обоих файлов должны совпадать; оба файла должны либо быть мб-файлами, либо принадлежать одному и тому же мд-контейнеру. В частности, $\phi1$ и $\phi2$ могут являться одним и тем же файлом. *

Операции пересылки данных. В случае непосредственного обмена мбмд-файлы рассматриваются как файлы с произвольным доступом. При этом считается, что элементы файла — это элементы его секторов. Адрес по файлу задается с точностью до сектора, т. е. адрес — это математический номер сектора относительно начала файла. Адресация с точностью до элемента невозможна. Количество считываемой или записываемой информации не обязательно должно быть кратно сектору. Однако обмен порциями, кратными сектору, позволяет более рационально использовать память на носителе.

Параметры операций. Параметрами операций синхронной пересылки данных являются: заголовок открытого фай-

ла (*ф*), адрес начала обмена (*а*) и массив обмена (*м*). Массив обмена может быть представлен:

1. Указателем с-вектора элементов формата ф64. В этом случае участок файла, считываемый или записываемый за одну операцию, не должен переходить с одного листа файла на другой.

*2. «Пачкой» указателей смежных векторов элементов формата ф64 (только в привилегированном режиме). В этом случае обмен производится последовательно для каждого из векторов. Параметр *м* в этом случае является целым числом, задающим адрес пачки. Участок файла, считываемый или записываемый в процессе обмена с каждым из векторов, не должен переходить с одного листа файла на другой.

В непривилегированном режиме накладываются следующие ограничения:

1. Вектор не может располагаться в стеке.

2. Один и тот же вектор не может одновременно участвовать в двух операциях обмена.

Нарушение ограничений приводит к возникновению «ошибки в параметрах обмена».

Ниже в квадратных скобках указывается форма вызова операций запуска параллельного обмена. Значением первого параметра в данном случае является блок ввода/вывода (*бвф*). К этому блоку предварительно должен быть прикреплен файл, над которым и будет осуществляться операция. Смысл остальных двух параметров совпадает с аналогичными параметрами синхронного обмена. Указатель семафора завершения обмена можно получить, запросив значение атрибута *смфобмена*. *

1. *чит* (*ф*, *а*, *м*) [*читпар* (*бвф*, *а*, *м*)]

В вектор *м* считывается содержимое последовательности элементов файла, начинающейся от начала сектора *а*. Количество считываемых элементов равно длине вектора *м*. Тип результирующих значений элементов вектора *м* определяется значением атрибута *типданных*.

В качестве результата выдается значение *ф* [*бвф*].

* Если адрес начала чтения приходится на математический лист файла, для которого еще не отведен физический лист, то возникает ошибка «нет листа».

Если адрес начала чтения выходит за пределы максимально возможного объема памяти файла, то возникает ошибка «физический конец файла». *

2. *зап* (*ф*, *а*, *м*) [*заппар* (*бвф*, *а*, *м*)]

Значения элементов вектора *м* переписываются в файл, в последовательность элементов, начинающуюся с сектора *а*. Количество заполненных элементов равно длине вектора *м*. Если оказывается, что последний сектор прописан не до конца, то-

его остаток заполняется нулями. В зависимости от значения атрибута *типданных* информация о типе записываемых значений сохраняется или теряется.

В качестве результата возвращается значение *ф* [*бвф*].

* Если адрес начала записи выходит за пределы максимально возможного объема файла, то возникает ошибка «физический конец файла». *

8.2. Общие свойства последовательного доступа. Ниже описываются файлы с последовательным доступом. Такой файл состоит из последовательности упорядоченных по номерам физических записей (зоны мл-файла, перфокарты пк-файла, строчки АЦПУ, группы символов пл-файла). Перед началом операции обмена головка записи/считывания находится между записями. Запись, предшествующая головке, т. е. только что обработанная (направление обработки — от начала файла к его концу), называется текущей. Номер текущей физической записи задается значением атрибута *блокфайла*. Каждая очередная операция обмена производит действия над записью, следующей за текущей. В результате выполнения операции текущая позиция перемещается. Соответственно корректируется значение атрибута *блокфайла*: при записи/чтении оно становится равным номеру записанной/считанной записи, а при позиционировании — номеру последней пропущенной записи. В число операций над файлами с последовательным доступом входят только такие операции, которые приводят к относительному смещению текущей позиции, а не к произвольной установке.

Параметрами операции синхронной пересылки данных являются заголовок открытого файла (*ф*) и указатель с-вектора байтовых элементов*) (*м*).

* Попытка чтения или позиционирования за пределами заполненной части файла приводит к ошибке «логический конец файла».

Если какая-либо операция, связанная с выводом информации, приводит к тому, что результирующий размер файла превышает ограничение, определенное атрибутом *максдлинфайла*, то возникает ошибка «физический конец файла».

В векторе *м* элемент с номером 0 должен быть прижат к началу слова памяти и весь вектор должен полностью располагаться на одной математической странице оперативной памяти (длина математической страницы ОЗУ — 512 слов). При операциях чтения мл-файла в обратном направлении последний

---

*) Следуя устоявшейся терминологии, в описании операций обмена будет использоваться термин «байтовый формат». Этот формат совпадает с литерным форматом и равен 8-ми разрядам.

элемент вектора должен быть прижат к концу слова памяти. Кроме того, в непривилегированном режиме на вектор накладываются те же ограничения, что и в случае обмена с мб- и мд-файлами. Длина вектора не обязательно должна быть кратна числу 8.

В случае параллельного обмена отличие структуры операции состоит в том, что первым параметром должен быть блок ввода/вывода (как и при произвольном доступе). Необходимо учитывать, что при параллельном обмене текущая позиция — это та, в которую головка записи/считывания установлена непосредственно перед началом реального выполнения внешним устройством операции пересылки данных или операции позиционирования, а не в момент инициирования операции программой. *

8.3. Мл-файлы. Записью мл-файла является зона. Длины различных зон одного файла могут отличаться. Зона состоит из последовательности байтовых элементов, упорядоченных по номерам. Количество элементов в зоне может изменяться от 18 до 999 999. Зоны отделяются одна от другой межзонным промежутком. Начальной зоне файла непосредственно предшествует начальная метка мл-файла, а за конечной зоной следует конечная метка. Эти метки представляют собой отображение заголовка мл-файла.

Текущая позиция характеризуется значением позиционного атрибута *блокфайла* (номер текущей зоны относительно начала файла). Нумерация зон мл-файла начинается с 1.

Для мл-файлов возможно позиционирование и чтение не только в прямом, но и в обратном направлении. В этом случае в соответствующей синтаксической конструкции указывается спецификация *пред*. Очередная операция осуществляется над записью, предшествующей головке записи/считывания, т. е., с учетом введенной выше терминологии,— над текущей записью.

Операции позиционирования.

1) *уст* (*ф*, *нач*) [*устпар* (*бвф*, *нач*)]

Текущая позиция устанавливается в положение, следующее за начальной меткой файла.

2) *уст* (*ф*, *кон*) [*устпар* (*бвф*, *кон*)]

Текущая позиция устанавливается в положение, предшествующее конечной метке файла.

3) *уст* (*ф*, *след* (*к*)) при $к \geqslant 1$ [*устпар* (*бвф*, *след* (*к*))]
уст (*ф*, *пред* (*к*)) [*устпар* (*бвф*, *пред* (*к*))]

Пропускается *к* зон файла.

4) *уст* (*ф*, *след*) [*устпар* (*бвф*, *след*)]
уст (*ф*, *пред*) [*устпар* (*бвф*, *пред*)]
уст (*ф*, *след*) эквивалентно *уст* (*ф*, *след* (1))
уст (*ф*, *пред*) эквивалентно *уст* (*ф*, *пред* (1))

Операции пересылки данных

1) *чит* (*ф*, *след*, *м*) [*читпар* (*бвф*, *след*, *м*)]
чит (*ф*, *пред*, *м*) [*читпар* (*бвф*, *пред*, *м*)]

Содержимое элементов зоны считывается в вектор *м*. Считывание в прямом направлении происходит в порядке возрастания номеров элементов зоны и вектора, начиная от 0. Считывание в обратном направлении происходит в порядке убывания номеров элементов, начиная от последних элементов зоны и вектора.

Если в векторе *м* меньше элементов, чем в зоне, то значением атрибута *переполнен* становится 1 и возникает ошибка «переполнение буфера». Если в векторе *м* больше элементов, чем в зоне, то значением этого атрибута становится —1, и вектор *м* заполняется не до конца (при обратном направлении — не до начала). Если число элементов в зоне и в векторе совпадают, то значение этого атрибута становится нулевым. Во всех случаях значением атрибута *длинблока* становится число, равное количеству элементов в зоне.

2) *зап* (*ф*, *след*, *м*) [*заппар* (*бвф*, *след*, *м*)]

Содержимое элементов вектора *м* записывается во вновь созданную зону файла. Длина этой зоны равна длине вектора *м*. Старая зона, которой соответствовал такой же номер (если такая была), и зоны с бо́льшими номерами (если такие были), а также все файлы, следующие за данным в контейнере (если такие были), ликвидируются. Новая зона считается последней зоной файла.

Если над файлом после операции записи выполняется какая-либо операция, отличная от записи, то этой последней операции предшествует запись на ленту конечной метки файла и конечной метки контейнера.

8.4. Зпк- и чпк-файлы. В последующем описании, если нет необходимости различать зпк- и чпк-файлы, употребляется термин пк-файлы.

Записью пк-файла является перфокарта. Перфокарта состоит из последовательности 12-разрядных элементов (колонок). Каждая перфокарта содержит 80 колонок. Начальной перфокарте пк-файла предшествует перфокарта, содержащая начальную метку пк-файла. Эта метка представляет собой отображение заголовка пк-файла. За конечной перфокартой памяти пк-файла следует перфокарта, содержащая конечную метку пк-файла. Структура меток пк-файла приведена в Приложении 4.

Текущая позиция характеризуется значением атрибута *блокфайла* (номер текущей перфокарты).

Операции позиционирования.

уст (ф, кон) [*устпар (бвф, кон)*]

Операция предназначена для чпк-файлов. Начиная с перфокарты, следующей за текущей, пропускаются все перфокарты чпк файла. Кроме того, пропускается перфокарта, содержащая конечную метку пк-файла.

Операции пересылки данных.

1) *чит(ф, след, м)* [*читпар(бвф, след, м)*]

Операция предназначена для чпк-файлов. Значения элементов перфокарты перекодируются и считываются в вектор *м*.

Если в векторе *м* меньше, чем 80 элементов (байтовых или 16-разрядных), то информация, находящаяся в старших колонках перфокарты, пропадает, и значение атрибута *переполнен* становится равным 1. Если в векторе *м* больше, чем 80 элементов, то вектор заполняется не до конца.

Возможны два варианта перекодировки:

А. Если значение атрибута *двоичный* есть ложь, то считается, что вектор *м* состоит из байтовых элементов, а информация на перфокарте представлена в коде КПК-12. Каждому 12-разрядному элементу перфокарты ставится в соответствие его 8-разрядное представление в коде ДКОИ-8. Это 8-разрядное представление и рассматривается как результат перекодировки. Если значение какого-либо 12-разрядного элемента перфокарты таково, что для него не существует соответствующего 8-разрядного эквивалента, то возникает «ошибка символа».

В. Если значение атрибута *двоичный* есть истина, то считается, что вектор *м* состоит из 16-разрядных элементов, а информация на перфокарте представлена в двоичном виде. Перекодировка значения элемента перфокарты состоит в том, что к левым и к правым 6 разрядам добавляется по два старших нулевых разряда. В результате образуется 16-разрядное значение, которое и рассматривается как результат перекодировки.

2) *зап (ф, след, м)* [*заппар (бвф, след, м)*]

Операция предназначена для зпк-файлов. Значения элементов вектора *м* (8- или 16-разрядные, в зависимости от атрибута *двоичный*) перекодируются в 12-разрядные значения. Последовательность полученных значений выводится в виде новой перфокарты файла.

Если в векторе *м* больше, чем 80 элементов, то используются только первые 80. Если в векторе *м* меньше, чем 80 элемен-

тов, то соответствующее количество колонок перфокарты со старшими номерами не будет содержать пробивок.

Возможны два варианта перекодировки:

А. Если значение атрибута *двоичный* есть ложь, то считается, что вектор *м* состоит из байтовых элементов, и значение каждого элемента перекодируется из кода ДКОИ-8 в 12-разрядное значение в коде КПК-12.

Б. Если значение атрибута *двоичный* есть истина, то считается, что вектор *м* состоит из 16 разрядных элементов. Перекодировка значения элемента вектора состоит в том, что в левых и правых разрядах значения убираются по 2 старших разряда. Образовавшееся 12-разрядное значение и рассматривается как результат перекодировки.

8.5. Ацпу-файлы. Записью ацпу-файла является строчка на бумажном носителе. Строчка состоит из последовательности литер, максимальная длина строки определяется значением атрибута *максдлинстроч*. Строчка печатается, начиная с самой левой печатной позиции носителя. Начальной строчке памяти файла предшествует начальная метка ацпу-файла, а за конечной строчкой следует конечная метка ацпу-файла. Эти метки представляют собой отображение заголовка файла.

* Текущая позиция характеризуется значением атрибута *блокфайла*. Если программа не осуществляет управление вертикальной табуляцией носителя (с помощью операции «прогон к каналу»), то значение этого атрибута в точности соответствует расположению печатающейся строчки относительно начала файла. В противном случае вертикальное расположение печатающейся строчки определяется программой.

Отличное от нуля значение атрибута *максдлинстран* задает размер логической страницы ацпу-файла. В этом случае система осуществляет следующие действия по разбиению текста на страницы. Если значение параметра *к* больше 0 (см. ниже операции позиционирования и пересылки данных), то при условии, что $ф'строчка < ф'максдлинстран$, в конце операции значение атрибута *строчка* корректируется: $ф'строчка = ф'строчка + к$; если же последнее условие не выполняется, то в конце операции атрибут *строчка* становится равным *к*, значение атрибута *страница* увеличивается на 1, и возникает ошибка «конец логической страницы». Операция «прогон к каналу» (см. операции позиционирования и передачи данных) не изменяет значение атрибута *строчка*, кроме случая прогона к каналу с номером 1. В этом случае значение атрибута *строчка* становится равно 1.

Замечание. Стандартная реакция системы на ошибку «конец логической страницы» не приводит к прогону бумаги к началу следующей физической страницы. При необходимости выравнивания границ логических и физических страниц надо, например, в переопределении стандартной реакции на ошибку

«конец логической страницы» предусмотреть табуляцию бумажного носителя с помощью явного вызова операций позиционирования. *

Операции позиционирования.

1) *уст* (*ф*, *след*(*к*)) при $к \geqslant 1$ [*устпар* (*бвф*, *след* (*к*)]

Выполняется вертикальная табуляция бумажного носителя на *к* интервалов, т. е. выводится $к - 1$ пустых строчек. Носитель устанавливается на текущую строчку.

2) *уст* (*ф*, *след*) [*устпар* (*бвф*, *след*)]

Операция эквивалентна *уст* (*ф*, *след* (1)).

3) *уст* (*ф*, *канал* (*к*)) [*устпар* (*бвф*, *канал* (*к*))]

Операция осуществляет вертикальную табуляцию бумажного носителя под управлением перфоленты. В результате носитель будет продернут до строчки, номер которой соответствует каналу с номером *к*. Например,

уст (*ф*, *канал* (1))

обычно реализует переход на начало следующей страницы бумажного носителя.

Операции пересылки данных. Семантика операций записи в ацпу-файл отличается от семантики аналогичных операций для других файлов тем, что установка новой позиции происходит после печати новой строчки, а не предшествует ей. Кроме того, используя непосредственный обмен, надо учитывать, что непосредственно после создания ацпу-файла текущая позиция установлена на первую строчку файла, а не предшествует ей.

1) *зап* (*ф*; *след* (*к*), *м*) при $к = 0$ [*заппар* (*бвф*, *след* (*к*), *м*)]

Выполнение операции состоит в следующем. Выводится текущая строчка с номером, равным текущему значению атрибута *блокфайла*. Значения элементов вектора *м* становятся значениями соответствующих элементов новой строчки. Если длина вектора *м* не превышает *ф'максдлинстроч* элементов, то длина строчки равна длине вектора *м*. Если вектор *м* содержит более *ф'максдлинстроч* элементов, то печатаются только первые *ф'максдлинстроч* литер.

Примечание. В результате выполнения этой операции текущая позиция не корректируется. Переход на новую строку достигается путем позиционирования, например:

уст(*зап*(*ф*, *след*(0), *м*), *след*).

Если позиционирования не сделано, то последующая печать будет происходить на той же строчке, опять начиная от самой левой печатной позиции.

2) *зап* (*ф*, *след* (*к*), *м*) при $к \geqslant 1$ [*заппар* (*бвф*, *след* (*к*), *м*)]
Операция эквивалентна *уст* (*зап* (*ф*, *след* (0), *м*), *след* (*к*)).

3) *зап* (*ф*, *след*, *м*) [*заппар* (*бвф*, *след*, *м*)]
Операция эквивалентна *зап* (*ф*, *след* (1), *м*).

4) *зап* (*ф*, *канал* (*к*), *м*) [*заппар* (*бвф*, *канал* (*к*), *м*)]
Операция эквивалентна *уст* (*зап* (*ф*, *след* (0), *м*), *канал* (*к*)).

8.6. Чпл- и зпл-файлы. В последующем описании, если нет необходимости различать чпл- и зпл-файлы, употребляется термин «пл-файл». Пл-файл представляет собой последовательность 8-разрядных элементов (или символов) на перфоленте.

Операция пересылки данных осуществляет чтение или запись группы элемеитов, начинающейся с текущего элемента. Если значение атрибута *кодблока* есть **истина**, то группы разделяются особым символом, не включаемым в группу. Этот символ (код конца блока) набирается из панели устройства.

Текущая позиция характеризуется значением атрибута *блокфайла*, который задает порядковый номер последней записанной или считанной группы символов.

Операции пересылки данных.

1) *чит* (*ф*, *след*, *м*) [*читпар* (*бвф*, *след*, *м*)].

Операция предназначена для чпл-файлов. Группа символов считывается в вектор *м*. Длина этой группы (*к*) определяется следующим образом:

— если значение атрибута *кодблока* есть **ложь**, то *к* равно длине вектора *м*;

— если значение атрибута *кодблока* есть **истина**, то *к* равно либо количеству элементов до ближайшего кода конца блока (не включая его), либо длине вектора *м* (если количество элементов до ближайшего кода конца блока больше либо равно длине вектора *м*). Головка считывания устаиавливается перед первым необработанным символом (код конца блока автоматически минуется).

* Если оказалось, что количество элементов до ближайшего кода конца блока (при значении атрибута *кодблока*, равном **истина**) больше либо равно длине вектора *м*, то значением атрибута *переполнен* становится 1, значением атрибута *длинблока* — длина вектора *м*, и возникает ошибка «переполнение буфера». В противном случае значением атрибута *переполнен* становится 1, вектор *м* заполняется не до конца, и значением атрибута *длинблока* становится длина заполненной части этого вектора. *

2) *зап* (*ф*, *след*, *м*) [*заппар* (*бвф*, *след*, *м*)]

Операция предназначена для зпл-файлов. Содержимое элементов вектора *м* выводится на перфоленту в виде очередной группы символов,

Если значение атрибута *кодблока* есть **истина**, то после группы вновь созданных элементов помещается символ кода конца блока.

8.7. Ацд- и пм-файлы. Записью ацд- и пм-файла является сообщение. Длины различных сообщений одного файла могут отличаться. Сообщение представляет собой последовательность литер. Различаются входные и выходные сообщения. Те сообщения, которые вводятся с клавиатуры устройства на носитель файла (экран ацд-файла, бумажный носитель пм-файла) и затем читаются программой, называются входными. Те сообщения, которые записываются на носитель файла программой, называются выходными.

Текущая позиция файла характеризуется значением атрибута *блокфайла* (номер текущего сообщения).

Особенностью файлов данного типа является то, что сообщения нумеруются не в порядке их появления на носителе, а в порядке операций чтения/записи, производимых программой. Ввиду того, что программа может произвести вывод сообщения на носитель даже при наличии еще не обработанной очереди входных сообщений, эти два способа нумерации в общем случае различаются.

Операции пересылки данных. Выходные сообщения записываются на носитель без какой-либо промежуточной буферизации. В отличие от этого при появлении входного сообщения происходит одно из двух:

— программа ожидает появления входного сообщения. В этом случае сообщение сразу же поступает в обработку;

— сообщение появляется до того, как программа его затребует с помощью операции чтения. В этом случае данное сообщение ставится последним в очередь входных сообщений данного файла (количество сообщений в этой очереди равно значению атрибута *очрдввода*). Кроме того, если значением атрибута *активность* является процедура, то эта процедура запускается как параллельный процесс.

Таким образом, с помощью указанного атрибута можно определить программную реакцию на независимую активность со стороны терминала.

Считается, что диалог человек-машина упорядочен некоторым соглашением, и входные и выходные сообщения не конкурируют между собой. Это значит, что если идет процесс ввода сообщения с клавиатуры на носитель устройства, то программа не может начать вывод своего сообщения на носитель до тех пор, пока первый процесс не кончится, и наоборот.

1) *чит*(*ф*, *след*, *м*) [*читпар*(*ввф*, *след*, *м*)]

При выполнении операции чтения может произойти одно из двух:

— очередь входных сообщений пуста. В этом случае выводится приглашение, если задан режим «с приглашением» (см. атрибут *приглашение*), и программа ожидает вводное сообщение. После появления текущего входного сообщения оно считывается в вектор *м*;

— очередь входных сообщений не пуста. В этом случае первое сообщение из очереди рассматривается как текущее, считывается в вектор *м* и удаляется из очереди входных сообщений.

Атрибуты *переполнен* и *длинблока* устанавливаются так же, как и для других последовательных файлов (см., например, мл-файлы).

2) *зап*(*ф*, *след*, *м*) [*заппар*(*бвф*, *след*, *м*)]

Содержимое вектора *м* выводится в качестве текущего выводного сообщения. Длина сообщения равна длине вектора *м*.

9. Буферизованный обмен

9.1. Блоки постоянной и переменной длины. Как указывалось выше, в операциях буферизованного обмена считается, что файл логически состоит из последовательности блоков. Элементам этой последовательности поставлены в соответствие номера. Различаются файлы с блоками постоянной длины и файлы с блоками переменной длины. В первом случае файл состоит из блоков, количество элементов в которых равно значению атрибута *максдлинблока*. Во втором случае файл состоит из блоков, количество элементов в которых переменно, но не превышает значения атрибута *максдлинблока*. Длина блока мб- и мд-файла измеряется в словах; на других устройствах длина блока задается в байтах.

Ниже для файлов всех типов приводится соответствие между блоками файла при буферизованном обмене и записями файла при непосредственном обмене.

М б - и м д - ф а й л ы являются файлами с блоками постоянной длины. Длина блока не должна превышать размера листа файла. Последовательность блоков файла максимально плотно располагается на последовательности математических листов файла в порядке возрастания номеров. Причем каждый блок полностью расположен на одном листе, а начало блока совпадает с началом сектора. Элементами блока являются элементы файла, попавшие в этот блок.

Мл-файлы являются файлами с блоками переменной длины. Значение атрибута *максдлинблока* не превышает 999 999 байтов. Физически блок с номером $к$ представляет собой зону с номером $к$. Элементами блока являются элементы зоны. Минимальная возможная длина блока составляет 18 байтов.

Зпк- и чпк-файлы являются файлами с блоками постоянной длины. Физически блоки зпк- и чпк-файлов располагаются на 80-колоночных перфокартах. Последовательность блоков памяти файла располагается на последовательности перфокарт таким образом, чтобы блок начинался с начала перфокарты. В общем случае блок может располагаться на $т$ перфокартах. $к$-му блоку соответствуют перфокарты с номерами от $т*к$ до $т*(к+1)-1$. Элементам блока соответствуют колонки перфокарт. Информация располагается на перфокарте, начиная с колонки с номером 0.

Если значение атрибута *двоичный* есть **ложь** (т. е. информация на перфокартах представлена в коде КПК-12), то 8-разрядные значения элементов блока представляют собой перекодированную форму соответствующих 12-разрядных значений элементов (колонок) перфокарты.

Если значение атрибута *двоичный* есть **истина** (т. е. информация на перфокартах представлена в двоичном виде), то блок должен содержать четное число байтов. Если считать, что блок состоит из 16-разрядных элементов, то 16-разрядные значения элементов блока представляют собой перекодированную форму (см. п. 8.4) соответствующих 12-разрядных значений элементов (колонок) перфокарты.

Ацпу-файлы являются файлами с блоками постоянной длины. Физически блок с номером $к$ представляет собой строчку файла с номером $к$. Элементами блока являются элементы строчки. Длина блока не должна превышать значения атрибута *максдлинстроч*, задающего реальную длину строчки на используемом устройстве.

Чпл-файлы со значением атрибута *кодблока*, равным значению **ложь**, являются файлами с блоками постоянной длины. Блок такого чпл-файла определяется следующим образом. Элементы чпл-файла последовательно объединяются в группы по $т$ элементов в группе, где $т$ определяется значением атрибута *максдлинблока*; $к$-я группа представляет собой блок с номером $к$. Элементами $к$-го блока являются элементы $к$-й группы.

Чпл-файлы со значением атрибута *кодблока*, равным значению **истина**, являются файлами с блоками переменной длины. Текущий блок такого файла представляет собой группу сим-

волов, участвующих в очередной операции считывания в буферный массив. Длина этой группы определяется, как и в операции чтения чпл-файла при непосредственном обмене (см. п. 8.6).

Зпл-файлы (независимо от значения атрибута *кодблока*) являются файлами с блоками переменной длины. Блоками такого файла являются группы символов, выведенные с помощью операций записи. После каждого очередного блока зпл-файла со значением атрибута *кодблока*, равным значению **истина**, помещается код конца блока.

Ацд- и пм-файлы являются файлами с блоками переменной длины. Физически блок с номером k представляет собой сообщение с номером k. Элементами блока являются литеры сообщения.

9.2. Буфер файла. Два способа буферизованного обмена. В процессе буферизованного обмена используются объекты, которые называются буферами данного файла. Основной компонентой буфера является буферный массив. Этот массив представляет собой с-вектор элементов, формат которых определяется значением атрибута *формэлем*. Длина вектора определяется значением атрибута *максдлинблока* файла. Значение атрибута *блокфайла* из буфера представляет собой целое число, равное номеру блока, которому соответствует данный буфер.

Кроме атрибута *блокфайла*, основными атрибутами буфера являются:

1) *длинблока*. Определяет реальную длину считанного или записываемого блока в случае файлов с блоками переменной длины. Для файлов с блоками постоянной длины значение этого атрибута равно значению атрибута *максдлинблока* из файла.

2) *модифицирован*. Определяет, запрашивался ли указатель для записи в буферный массив.

* Когда задача производит чтение или запись некоторого блока файла, создается буфер, соответствующий этому блоку (если такого буфера еще нет). Этот буфер прикрепляется к задаче (или число ссылок на существующий буфер увеличивается на 1). В связи с тем, что могут одновременно существовать несколько путей доступа к мбмд-файлу, один и тот же буфер мбмд-файла может быть одновременно прикреплен к нескольким задачам. Более того, один и тот же буфер мбмд-файла может быть одновременно прикреплен с помощью нескольких ссылок к одной задаче. Таким образом, если в данный момент существует несколько путей буферизованного доступа к одному и тому же блоку некоторого мбмд-файла, то все они осуществляются через один и тот же буфер. *

Операция *чит* считывает содержимое блока в буферный массив и выдает указатель, описывающий этот массив. Операция

зап подготавливает буфер, предназначенный для последующей записи в файл, и выдает указатель, описывающий буферный массив. Если предполагается сделать запись в заполненную часть файла, то при подготовке буфера в него считывается текущее содержимое блока. Используя полученный указатель, программа может заполнять (или модифицировать) буферный массив. Фактический сброс его содержимого в файл происходит неявно после того, как программа закончила обработку данного блока (точнее — обработку соответствующего буфера).

Операция *чит* выдает указатель, описывающий часть буферного массива от элемента с номером 0 до элемента с номером *буф'длинблока* — 1. Пользуясь этим указателем, можно только считать значения элементов массива.

Попытка записи приводит к возникновению ситуации *нарушениезащиты*.

Операция *зап* выдает указатель, описывающий весь буферный массив (размер предварительно считанного блока можно узнать, запросив значение атрибута *длинблока*). С помощью этого указателя можно считывать значения элементов массива и изменять их.

* *Два способа обмена.* Различаются два способа буферизованного обмена: однобуферный и многобуферный. В первом случае с файлом связана позиционная переменная, которая указывает на текущий обрабатываемый блок файла. Когда позиционная переменная смещается на другой блок, происходит автоматическое открепление буфера, соответствующего предыдущему обрабатываемому блоку. Во втором случае такой специальной позиционной переменной не существует. Позиция указывается с помощью номера блока. Начало обработки некоторого блока не приводит к автоматическому откреплению буфера, соответствующего предыдущему обрабатываемому блоку. Открепление буфера осуществляется только явным образом. Таким образом, в этом способе имеется возможность иметь одновременно несколько прикрепленных буферов и тем самым работать одновременно с несколькими блоками файла.

Запись блока. Если в результате открепления буфера (*буф*) оказывается, что на данный буфер не осталось временных ссылок, то такой буфер ликвидируется. Перед ликвидацией проверяется значение атрибута *модифицирован*. Если оно есть **истина**, то происходит сброс содержимого буферного массива на файл. Это означает, что в файл записывается блок с номером *буф'блокфайла*. Длина этого блока определяется значением атрибута *буф'длинблока*. Старый блок с таким же номером (если он есть) ликвидируется. Кроме того, в случае мл-файла ликвидируются блоки с большими номерами (если такие есть) и файлы, расположенные следом за текущим в мл-контейнере (если такие есть). Запись буферного массива в файл выполняется так же, как запись массива при непосредственном обмене. Особенности записи, связанные с типом устройства, см. в § 8.

Ошибки. В начале выполнения операции чтения или записи (в обоих способах буферизованного обмена) производятся следующие проверки логических и физических границ. Если при чтении оказывается, что необходимо считать блок, находящийся за пределами заполненной части файла, то возникает ошибка «логический конец файла». Кроме того, если при чтении мбмд-файла оказывается, что необходимо считать блок, находящийся на математическом листе, для которого не распределен соответствующий физический лист, то возникает ошибка «нет листа». Если в начале выполнения операции записи оказывается, что предполагается сделать запись блока, находящегося за пределами максимально возможного объема файла, то возникает ошибка «физический конец файла».

Размер заполненной части файла определяется системой следующим образом. В случае мб- и мд-файлов он задается значением атрибута *блоклкф*, который равен номеру первого незаполненного блока. На других устройствах конец заполненной части файла определяется по характерным физическим признакам носителя: конечная метка мл-файла, конечная перфокарта пк-файла, конец пробивок на перфоленте. В дальнейшем изложении для обозначения номера первого незаполненного блока будет использоваться идентификатор *блкф*.

При записи нового блока в файл может оказаться, что его номер больше *блкф*, т. е. между последним блоком заполненной на текущий момент части файла и новым блоком существуют незаполненные блоки. В этом случае предпринимаются следующие действия:

— мб-, мд-файл. Незаполненные блоки, так же как и новый блок, включаются в заполненную часть файла, т. е. значение атрибута *блоклкф* становится равно: номер нового блока +1. Первоначальное содержимое незаполненных блоков не определено;

— ацпу-файл. Незаполненные блоки выводятся на носитель в виде пустых строчек;

— файлы остальных типов. Возникает ошибка «логический конец файла». *

10. Однобуферный способ

Напомним, что в этом способе обмена считается, что существует текущая позиция, и номер текущего блока хранится в позиционной переменной. Ниже в описании семантики операций встречается присваивание атрибуту *блокфайла*. Такая форма используется в качестве условного обозначения для описания смещения текущей позиции.

Позиционная переменная имеет дополнительный позиционный атрибут *элемблока*. Программа может придавать этому атрибуту целочисленные значения, уточняя тем самым положение текущей позиции с точностью до элемента текущего блока (см. операции 7, 8 п. 10.2).

Для мб-, мд- и мл-файлов разрешено позиционирование и чтение не только в прямом, но и в обратном направлении. В этом случае в соответствующей синтаксической конструкции указывается спецификация *пред*. Соответствующие этому случаю уточнения в описании семантики операций заключены в квадратные скобки.

* Для ацпу-файла значение атрибута *максдлинстран*, отличное от нуля, задает размер логической страницы. В этом случае семантика ошибки «конец логической страницы» и изменения атрибута *пзп'строчка* и *пзп'страница* аналогична той, которая описана для случая непосредственного обмена.

Одновременно несколько позиционных переменных могут существовать только для мб-, мд-, ацд-, пм-файла, причем только в случае мб- и мд-файла они могут работать через один и тот же буфер. В силу этого ограничения замечания в п. 9.2, связанные с одновременной работой нескольких позиционных переменных через один и тот же буфер, имеют отношение только к мб- и мд-файлам. *

10.1. Операции позиционирования.

1) *уст*(*пзп*,*нач*)

Операция предназначена для мб-, мд- и мл-файлов. Текущая позиция перемещается в начало файла. Результат: *пзп*.

2) *уст*(*пзп*,*кон*)

Операция предназначена для мб-, мд-, мл-, чпл-, чпк-файлов. Текущая позиция перемещается на конец файла. Результат: *пзп*.

3) *уст*(*пзп*,*след*(*к*)) при $к \geqslant 1$
[*уст*(*пзп*,*пред*(*к*))]

Операция предназначена для ацпу-файлов и производит следующие действия:

пзп'блокфайла := *пзп'блокфайла* + *к*
[*пзп'блокфайла* := *пзп'блокфайла* — *к*]
Результат: *пзп*.

4) *уст*(*пзп*,*след*) [*уст*(*пзп*,*пред*)]

Операция предназначена для ацпу-файлов и эквивалентна:
уст(*пзп*,*след*(1)) [*уст*(*пзп*,*пред*(1))]

5) *уст*(*пзп*,*канал*(*к*)) при $к \geqslant 0$

Дополнительная операция для ацпу-файлов, связанная с вертикальной табуляцией бумажного носителя под управлением перфоленты.

10.2. Операции пересылки данных. Последовательный доступ.

1) *чит*(*пзп*,*след*(*к*)) при $к \geqslant 1$
чит(*пзп*,*пред*(*к*))

Операция предназначена для мб-, мд-, мл-, чпк-, чпл-, ацд- и пм-файлов. Для ацд- и пм-файлов значение *к* должно быть равно 1.

Операция перемещает текущую позицию и выдает указатель с-вектора. Этот указатель описывает буферный массив, содержащий блок, который стал текущим.

Перемещение позиции происходит следующим образом. Если текущая позиция была установлена на начало файла, то

пзп'блокфайла := *к* — 1

а иначе

пзп'блокфайла := *пзп'блокфайла* + *к*,

[Если позиционная переменная установлена на конец файла, то

пзп'блокфайла := *блкф* — *к*,

а иначе

пзп'блокфайла := *пзп'блокфайла* — *к*.]

В результате выполнения операции атрибут *элемблока* принимает значение 0.

2) *чит*(*пзп*,*след*) [*чит*(*пзп*,*пред*)]

Операция эквивалентна:

чит(*пзп*,*след*(1)) [*чит*(*пзп*,*пред*(1))]

3) *зап*(*пзп*,*след*(*к*)) при *к* ⩾ 1

Операция предназначена для мб-, мд-, мл-, зпк-, апцу-, зпл-, апд- и пм-файлов. Для зпк- и зпл-файлов значение *к* должно быть равно 1.

Операция перемещает текущую позицию и выдает указатель с-вектора. Этот указатель описывает буферный массив, предназначенный для записи блока, ставшего текущим.

Перемещение позиции происходит следующим образом. Если текущая позиция была установлена на начало файла, то

пзп'блокфайла := *к* — 1

Если текущая позиция была установлена на конец файла, то

пзп'блокфайла := *блкф* + *к* — 1

В остальных случаях

пзп'блокфайла := *пзп'блокфайла* + *к*

В результате выполнения операции атрибут *элемблока* принимает значение 0.

* Примечание. Завершение выполнения операции *зап* не приводит к сбросу содержимого буферного массива на файл. Как указано в п. 9.2, сброс происходит лишь при полном откреплении буфера. Однако имеет место важный частный слу-

чай. Если файл прикреплен только к одной позиционной переменной, то выполнение операции *зап* приводит к тому, что перед созданием буфера для следующего блока происходит полное открепление предыдущего буфера и его сброс (параллельно с выполнением программы). *

4) *зап*(*пзп*,*след*) Операция эквивалентна *зап*(*пзп*,*след*(1)).

5) *нов*(*пзп*,*след*(*к*))

Эта операция отличается от описанной выше операции записи тем, что первоначальное содержимое буферного массива не определено. Ее использование позволяет избежать считывания блока файла в буферный массив, если в этом нет необходимости.

6) *нов*(*пзп*,*след*)

Операция эквивалентна *нов*(*пзп*,*след*(1)).

7) *чит*(*пзп*,*след*(*к*)) при $к = 0$ [*чит*(*пзп*,*пред*(*к*))]

К моменту начала исполнения этой операции блок *пзп'блок-файла* либо уже считан в соответствующий буфер, либо процесс считывания начат, но еще не окончен. В последнем случае операция ожидает завершения считывания. В качестве результата операции выдается указатель для считывания, описывающий часть буферного массива, начинающуюся с элемента с номером *пзп'элемблока* [кончающуюся элементом с номером *пзп'элемблока*]. Если текущая позиция была установлена на начало файла [конец файла], то операция эквивалентна *чит*(*пзп*,*след*(1)) [*чит*(*пзп*,*пред*(1))]

8) *зап*(*пзп*,*след*(*к*)) при $к = 0$

Данная операция отличается от предыдущей тем, что выдает указатель для записи/считывания. Кроме того, значением атрибута *буф'модифицирован* становится **истина**.

* **10.3. Произвольно-последовательный доступ.** Описываемые ниже операции предназначены в основном для использования в режиме произвольно-последовательного доступа к файлу, т. е. когда производится несколько серий последовательной обработки, но каждой серии предшествует установка текущей позиции в новое положение. Операции устанавливают текущую позицию на блок с заданным номером и осуществляют, если надо, его считывание с файла. Все операции определены только для мб-, мд- и мл-файлов.

Операции чтения и записи могут выполняться как в синхронном, так и в параллельном режиме. Особенности выполнения параллельного режима аналогичны тем, которые отмечены для многобуферного обмена (см. § 11).

Операции пересылки данных.

чит(*пзп*,*нбл*) [*читпар*(*пзп*, *нбл*)]
зап(*пзп*,*нбл*) [*заппар*(*пзп*,*нбл*)]
нов(*пзп*,*нбл*)

Данные операции отличаются от соответствующих операций последовательного доступа тем, что все действия производятся для блока с номером *нбл*. Текущая позиция перемещается на этот блок, и он становится текущим.

Пример. Синхронизация с помощью операции *чит*.

```
текуказ := читпар (пзп , текблок) [текэлем :] ;
... предварительные действия ... ;
чит (пзп , след(0)) ;          % синхронизация
а <:= текуказ длиной к % обработка считанного блока.
```

Пример. Синхронизация с помощью операции *ждать*.

```
ждать (читатр (пзп,смфобмена))
```

• 11. Многобуферный способ

Данный способ буферизованного обмена определен только для мб- и мд-файлов. Значением первого параметра во всех операциях должна быть таблица буферов.

Операции произвольного доступа могут осуществляться как в синхронном, так и в параллельном режиме. Параллельные операции произвольного доступа отличаются от синхронных тем, что только инициируют считывание блока с номером *нбл* в буферный массив (если выполняется операция *заппар* и блок с номером *нбл* находится за пределами заполненной части файла, то, как и в случае других операций записи, предварительное считывание не происходит). На этом исполнение операции заканчивается. Процесс реального считывания блока происходит параллельно с выполнением программы. Эти операции используются в тех случаях, когда необходимо, чтобы процесс считывания происходил на фоне операций, предшествующих началу обработки буферного массива. В качестве значения эти операции выдают соответственно указатель для записи/считывания из буферного массива. Следует учитывать, что перед тем как использовать этот указатель для обращения к элементам буферного массива, необходимо выполнить какую-либо синхронизующую операцию, например:

```
буф :=.читпар(тбф , нбл) ;
... предварительные действия ...
чит(тбф , нбл) ;
а <:= буф длиной к
```

или явный способ синхронизации:

```
ждать (читатр (тбф,буф,смфобмена))
```

Операции явного сброса буферов также могут осуществляться как в синхронном, так и в параллельном режиме. Параллельная операция сброса буферов только инициирует процесс записи содержимого буферного массива на файл. Затем управление возвращается в программу, и процесс записи происходит параллельно с выполнением программы.

Форма конструкций параллельного обмена приведена в квадратных скобках рядом с формой для синхронного обмена.

11.1. Произвольный доступ.

1) *чит(тбф,нбл)* [*читпар(тбф,нбл)*]

В качестве результата операция выдает указатель, который описывает буферный массив, содержащий блок *нбл*. С помощью полученного указателя можно только считывать элементы буферного массива. Попытка записи приводит к возникновению ситуации *нарушениезащиты*.

2) *зап(тбф,нбл)* [*заппар(тбф,нбл)*]

Операция в качестве результата выдает указатель, который описывает буферный массив, предназначенный для записи блока *нбл*. С помощью этого указателя можно записывать значения в буферный массив и считывать значения из него.

3) *нов(тбф,нбл)*

Эта операция отличается от *зап(тбф,нбл)* тем, что первоначальное содержимое буферного массива всегда неопределено. Ее использование позволяет избежать считывания блока памяти файла в буферный массив, если в этом нет необходимости.

11.2. Последовательный доступ. В пп. 1—8 семантика операций последовательного доступа выражена через ранее определенные операции. Форма эквивалентной операции приведена справа от знака равенства.

1) *чит(тбф,буф,след(к))* при $к \geqslant 1$
$= чит(тбф,буф'блокфайла + к)$

2) *чит(тбф,буф,след)*
$= чит(тбф,буф,след(1))$

3) *зап(тбф,буф,след(к))* при $к \geqslant 1$
$= зап(тбф,буф'блокфайла + к)$

4) *зап(тбф,буф,след)*
$= зап(тбф,буф,след(1))$

5) *нов(тбф,буф,след(к))* при $к \geqslant 1$
$= нов(тбф,буф'блокфайла + к)$

6) *нов(тбф,буф,след)*
$= нов(тбф,буф,след(1))$

7) *чит(тбф,буф,пред(к))*
$= чит(тбф,буф\ блокфайла - к)$

8) *чит(тбф,буф,пред)*
$= чит(тбф,буф,пред(1))$

9) *уст(тбф,нач)*

В качестве результата операции выдается объект типа указатель с-вектора. Этот указатель не описывает какой-либо массив, но может быть использован для того, чтобы идентифицировать позицию «начало файла». В семантике операций пп. 1—6 предполагается, что в этом случае

$буф'блокфайла = -1$

10) *уст(тбф,кон)*

В качестве результата операции выдается объект типа указатель. Этот объект не описывает какой-либо массив, но может быть использован для того, чтобы идентифицировать позицию «конец файла». В семантике операций пп. 1—6 предполагается, что в этом случае

буф'блокфайла = *ф'блоклкф* — 1
а в операциях пп. 7, 8
буф'блокфайла = *ф'блоклф*

11.3. Операции над буферами.

1) *зап*(*тбф*,*буф*,*след*(*к*)) при *к* = 0

Выполнение операции состоит в том, что значением атрибута *буф'модифицирован* становится **истина**. В качестве результата операции выдается указатель *буф* с открытым разрешением записи и считывания.

2) *открепбуф*(*тбф*,*буф*1,*буф*2, ... *буфк*)

Операция осуществляет открепление буфера *буф*1 от данной задачи (при этом уничтожается только одна ссылка из задачи на этот буфер). Если теперь оказывается, что данный буфер не прикреплен ни к одной задаче, то соответствие между буфером и блоком файла ликвидируется. При этом, если значение *буф*1'*модифицирован* есть **истина**, то осуществляется сброс содержимого буферного массива. Здесь следует сделать уточнение относительно открепления буфера. Буфер, в отличие от других объектов, при полном откреплении не уничтожается; система может назначить данный буфер для другого блока. Для уничтожения буфера служит операция *ликвидбуф*.

Аналогичные действия осуществляются с буферами *буф*2, ..., ..., *буфк*.

3) *открепбуф*(*тбф*)

Операция осуществляет полное открепление всех буферов файла от данной задачи с уничтожением всех ссылок из данной задачи на эти буфера. Как и в предыдущем случае, это может привести к сбросу содержимого буферных массивов, но, вообще говоря, не означает ликвидацию буферов.

4) *ликвидбуф*(*тбф*, *буф*1, ..., *буфк*)

Операция осуществляет полное открепление буфера *буф*1 от данной задачи с уничтожением всех ссылок из данной задачи на этот буфер. Если теперь оказывается, что буфер не прикреплен ни к одной задаче, то он уничтожается. При этом, если значение атрибута *буф*1'*модифицирован* есть **истина**, то осуществляется сброс содержимого буферного массива.

Аналогично обрабатываются буфера *буф*2, ..., *буфк*.

5) *ликвидбуф*(*тбф*)

Операция осуществляет операцию п. 4) над всеми буферами файла, прикрепленными к данной задаче.

6) *сброс*(*тбф*,*буф*1,*буф*2, ..., *буфк*)
[*сброспар*(*тбф*,*буф*1,*буф*2, ..., *буфк*)].

Операция осуществляет сброс содержимого буферных массивов буферов *буф*1, *буф*2, ..., *буфк* в файл. Буфера после сброса не открепляются; соответствие между буферами и связанными с ними блоками продолжает сохраняться. Таким образом, данная операция дает возможность при необходимости приводить в соответствие содержимое буферных массивов и содержимое блоков на внешнем носителе. Операция может осуществляться как в синхронном, так и в параллельном режиме.

7) *сброс*(*тбф*) [*сброспар*(*тбф*)]

Операция осуществляет сброс содержимого всех буферных массивов на файл (подробно см. предыдущую операцию).

12. Структура текстового файла

В этом разделе описывается структура одного из встроенных типов файлов — текстового файла. Для такого файла значение атрибута *типфайла* равно числу 3.

Текстовый файл представляет собой обычный файл с присущим ему типом носителя, на котором он расположен, и прочими атрибутами, определяющими его характеристики. Особенность текстового файла состоит в том, что вводится ряд соглашений относительно структуры содержимого файла и ряд специфических атрибутов, характеризующих эту структуру.

Структура строки. Считается, что текстовой файл состоит из строк. Если значение атрибута *длинстрок* отлично от нуля, то длина всех строк одинакова и равна значению этого атрибута. В противном случае файл состоит из строк различной длины. Длина строки в обоих случаях не должна превышать значение атрибута *максдлинблока*. Строка представляет собой последовательность байтовых элементов. Строка может состоять из следующих компонент:

— фиксированный номер строки. Наличие этой компоненты зависит от значения атрибута *длинфнс* (см. ниже);

— информационная часть строки;

— байт, кодирующий конец строки, или байт, содержащий длину строки (в зависимости от типа внешнего устройства, см. ниже).

Нумерация строк. Различаются нумерованные текстовые файлы (значение атрибута *длинфнс* отлично от нуля) и ненумерованные текстовые файлы (значение атрибута *длинфнс* равно нулю). Каждая строка нумерованного файла содержит фиксированный номер этой строки. Фиксированный номер является целым числом, количество цифр в котором определяется значением атрибута *длинфнс* (в последующем описании, если не оговорено особо, под номером строки понимается ее фиксированный номер). Текстовый файл не может содержать двух разных строк с одинаковыми номерами. Строки расположены строго в порядке возрастания номеров, причем номера соседних строк могут отличаться более чем на 1.

Строка ненумерованного файла не содержит своего номера. При распечатке таких файлов применяется сквозная (через 1) нумерация строк. В результате при вставке или удалении строк старая нумерация строк постоянно меняется.

Структура файла. Расположение строк в памяти файла зависит от типа внешнего устройства.

В мб- и мд-файлах каждый блок файла содержит некоторое число полных строк (т. е. строка не переходит с одного блока на следующий). Строки располагаются последовательно, друг за другом. Первый байт строки содержит ее длину. Далее следуют байты, содержащие номер строки (для нумерованных файлов). Если значение атрибута *кодфнс* равно 0, то номер закодирован в двоичном виде, и под него отводится столько байтов, сколько необходимо, чтобы уместить максимальный номер (определяемый значением атрибута *длинфнс*). Если значение атрибута *кодфнс* равно 1, то номер закодирован в литерном виде, и под него отводится *длинфнс* байтов. За номером

следуют информационные байты строки. Первый байт строки содержит ее полную длину, т. е. количество байтов, отведенных под номер, плюс длина информационной части, плюс 1. За последней строкой блока следует байт со значением, равным нулю («забой»).

В отличие от мб- и мд-файла мл-файл является файлом с блоками переменной длины. Структура строки и расположение строк внутри блока строятся по аналогии с мб- и мд-файлами.

В пк-файлах каждая новая строка начинается с левой крайней колонки карты. Строка может занимать одну или несколько перфокарт. Строки текстового файла с плавающей длиной строки должны заканчиваться байтом «конец строки». Номера строк в файле с фиксированной нумерацией кодируются всегда в литерном виде и занимают *длинфнс* колонок. В отличие от мб- и мд-файлов номер строки не обязательно должен располагаться в начале строки. В общем случае позиция номера задается значением атрибута *позфнс*.

Пл-файл состоит из последовательности строк, расположенных друг за другом. Строки текстового файла с плавающей длиной строки должны заканчиваться байтом «конец строки». Номера строк в нумерованном файле кодируются всегда в литерном виде, занимают *длинфнс* символов и располагаются в начале строки. Если значение атрибута *кодблока* есть **истина**, то группы строк могут отделяться кодом конца блока (см. атрибут *кодблока*).

Сообщение текстового ацд- и пм-файла содержит некоторое число строк. Строка не может переходить из одного сообщения в следующее. Строка должна заканчиваться байтом «конец строки». Сообщение, вводимое в режиме «с приглашением в виде номера строки», должно содержать одну строку.

Значение атрибута *длинфнс*, отличное от нуля, указывает на то, что вводная строка должна содержать свой номер. Этот номер располагается в строке, начиная с позиций *позфнс*, и содержит *длинфнс* символов. Номер может попасть в строку двумя способами:

1. При чтении в режиме «с приглашением в виде номера строки» сама система помещает в строку ее номер.

2. В других режимах необходимо при вводе сообщения с клавиатуры поместить в каждую строку сообщения ее номер.

Контекст файла. С текстовым файлом, так же как и с файлом объектного кода, можно связать некоторый справочник (см. атрибут *внешконт*). Такой справочник является корневым справочником внешнего контекста текстового файла. В этом контексте могут находиться объекты, ссылки на которые тем или иным образом обозначены в тексте файла или закодированы в его атрибутах (см. ниже атрибут *имяяз*).

Имя языка. Содержимое текстового файла может представлять собой текст программы на некотором языке программирования. В этом случае атрибут *имяяз* содержит строку литер (S), представляющую собой имя программы транслятора данного языка программирования. Если предположить, что X — это указатель, установленный на корневой справочник внешнего контекста текстового файла, то ⟨внешнее имя⟩ X//S приводит к нужному транслятору. Одно из использований этого

атрибута связано с неявной трансляцией текста, происходящей в том случае, если в конструкции ⟨вызов⟩ вместо процедуры используется текстовый файл (см. п. 2.1 этой главы и § 12 гл. 2). *

13. Изображение файла

В программе файл может появиться следующими способами:

1. Программа может создать новый файл с помощью генератора.

2. Программе, как фактический параметр, может быть передан существующий файл.

3. Программа может обратиться к существующему файлу с помощью ⟨внешнего имени⟩.

4. Файл может быть непосредственно изображен в тексте программы аналогично числовым константам и массивам констант. Такой файл называется константным файлом. Константный файл изображается в программе с помощью конструкции «изображение файла».

⟨константный файл⟩ ::=
⟨изображение произвольного файла⟩
| ⟨изображение текстового файла⟩
⟨изображение произвольного файла⟩ ::=
файл ⟨вложенный файл⟩
⟨изображение текстового файла⟩ ::=
тфайл{(⟨список установок атрибутов⟩)}
⟨определение ограничителя⟩
⟨текст файла⟩⟨ограничитель⟩

13.1. Произвольный файл. ⟨Изображение произвольного файла⟩ может присутствовать только в программах, текст которых является пк-файлом (т. е. текст программы расположен на перфокартах или на другом носителе, например, диске или барабане, на котором в данном случае он должен быть представлен в виде псевдо-чпк-файла).

⟨Вложенный файл⟩ является самостоятельным пк-файлом, имеющим обычную структуру. Он представляет собой последовательность перфокарт, начинающуюся с начальной метки и кончающуюся конечной меткой. В начальной метке вложенного пк-файла определяются его атрибуты. Значения этих атрибутов могут не совпадать со значениями аналогичных атрибутов охватывающего пк-файла (в частности, в текстовый файл может быть вложен двоичный файл). Перфокарты, находящиеся между начальной и конечной меткой, определяют содержимое изображенного файла.

Внутри текстового ⟨вложенного файла⟩ могут опять встречаться ⟨изображения произвольного файла⟩.

Результатом выполнения ⟨изображения произвольного файла⟩ является постоянный указатель внешнего объекта, установленный на данный чпк-файл. Атрибуты файла совпадают с атрибутами, указанными в начальной метке ⟨вложенного файла⟩.

Обычно текст программы с ⟨изображениями произвольных файлов⟩ образуется следующим образом. Отдельно пробивается колода с основным текстом программы (например, текст за-

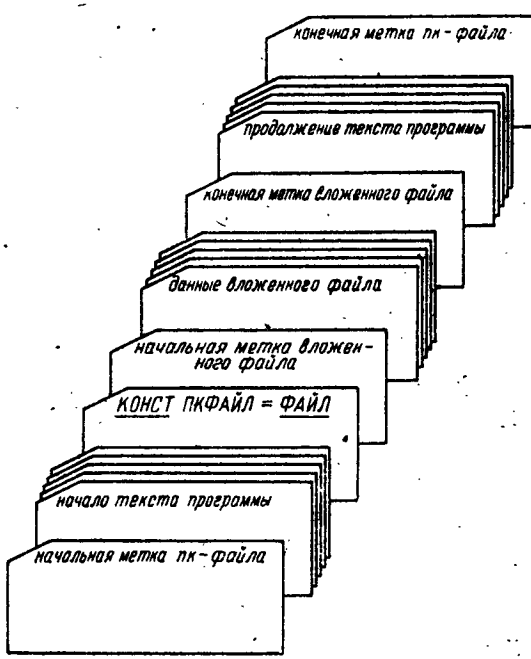


Рис. 3.1. Программа с вложенными файлами.

дания пакетного режима) и отдельно — колоды ⟨вложенных файлов⟩. Затем последние вкладываются в нужные места основной колоды (см. рис. 3.1).

13.2. Текстовый файл. Конструкция ⟨изображение текстового файла⟩ предназначена для изображения в программе константных файлов, содержащих произвольную текстовую информацию, в частности, текст автокод-программ, текст программ на других языках программирования. Константный текстовый

файл может быть изображен и с помощью описанного выше ⟨изображения произвольного файла⟩. Однако имеются следующие отличия:

— текст охватывающей программы не обязательно должен быть расположен на перфокартах;

— не требуется особой пробивки для начальной и конечной метки;

— текст файла из ⟨изображения текстового файла⟩ не является самостоятельным файлом в том смысле, что большинство его атрибутов, в частности, структура строк, определяются аналогичными атрибутами охватывающего файла.

Ограничитель текста. В ⟨определении ограничителя⟩ задается закрывающий ограничитель текста файла. Определение ограничителя отыскивается следующим образом. После служебного слова **тфайл** пропускаются (если они есть) пробелы, комментарии и ⟨список установок атрибутов⟩ в круглых скобках. Обозначим литеру, следующую за пропущенными, через X (в приведенном ниже примере это литера *). Тогда ограничителем файла является последовательность литер, расположенная между литерой X и следующим вхождением той же литеры (в приведенном примере последовательность представляет собой два восклицательных знака).

Пример.

```
тфайл
% на следующей строке определен ограничитель !!
*!!*
этот текст из 3 строк
является содержимым
текстового файла
!!
```

Литера X не может быть пробелом, символом процента и левой круглой скобкой.

Структура текста. Текст файла представляет собой последовательность строк. Первой строкой этой последовательности является строка, следующая за той, на которой определен закрывающий ограничитель файла, а последней строкой является строка, предшествующая той, на которой найдено вхождение закрывающего ограничителя файла.

Атрибуты файла. Результатом выполнения ⟨изображения текстового файла⟩ является постоянный указатель, установленный на данный текстовый мб- или мд-файл. Полученный файл имеет следующие атрибуты:

— *типфайла* = 3. Если текст написан на некотором языке программирования, то этот язык можно указать, задав в ⟨списке установок атрибутов⟩ атрибут *имяяз*;

— *пакстрок* и *длинстрок* совпадают с аналогичными атрибутами текстового файла охватывающей программы;

— *блоклкф* равен номеру последнего заполненного блока плюс 1. Атрибуты *максдлинблока* и *длинлиста* устанавливаются транслятором.

В ⟨списке установок атрибутов⟩, кроме прочих, можно указать атрибуты *длинфнс*, *кодфнс*, *фнс*, *шагфнс*. Тогда в соответствии со значениями этих атрибутов в начало каждой строки будет помещен ее номер, и в результате образуется нумерованный файл. Если эти атрибуты не указаны в ⟨списке установок атрибутов⟩, то независимо от нумерации охватывающего файла полученный файл является ненумерованным файлом.

Если внешний контекст текстового файла явно не задан (см. атрибут *внешконт*), то контекстом файла становится внешний контекст программы, в тексте которой изображен данный файл.

14. Константный справочник

⟨Изображение справочника⟩ предназначено для обозначения константного справочника. Константный справочник представляет собой оптимизированный вариант обычного архивного справочника. Оптимизация достигается в следующих моментах:

— константный справочник транслируется, а не создается специальными операциями;

— константный справочник хранится в компактной форме в файле объектного кода программы;

— доступ к элементам константного справочника реализуется более эффективными способами, чем в случае обычных справочников.

⟨изображение справочника⟩ ::=
спрконст (⟨список элементов справочника⟩)
⟨элемент справочника⟩ ::= ⟨идентификатор⟩{@} =
⟨выражение⟩

Результатом выполнения ⟨изображения справочника⟩ является постоянный указатель внешнего объекта, установленный на данный константный справочник. Этот справочник состоит из элементов, перечисленных в ⟨списке элементов справочника⟩. Имя каждого элемента задается компонентой ⟨идентификатор⟩, а значение элемента — компонентой ⟨выражение⟩.

В отличие от обычного справочника элемент константного справочника может содержать значения любого типа.

Использование. Константные справочники можно использовать в тех же операциях и конструкциях, что и обычные справочники. Исключение составляют операции создания элемента справочника и записи в элемент справочника, которые в данном случае запрещены.

Особенности выполнения. Существует различие в семантике выполнения операций над обычными и над константными справочниками. В случае константных справочников семантика операций и конструкций, использующих значение элемента справочника, имеет следующую особенность: значение элемента здесь определяется как результат выполнения соответствующего ⟨выражения⟩; выполнение происходит каждый раз при обращении к содержимому элемента.

Значение элемента справочника может приводить к внешнему объекту или к элементу другого справочника. Такой элемент константного справочника эквивалентен элементу обычного справочника, содержащему эквивалентную ссылку. Его можно использовать обычным образом, в частности,— как промежуточное звено при поиске по внешнему имени.

Для того чтобы получить непосредственно значение элемента константного справочника, надо воспользоваться конструкцией ⟨генератор объекта связи⟩ со спецификацией *прогр* (см. последнее присваивание в примере 1).

Пример 1.

```
начало
    ф64 локсправ , файл1 , прог1 ;
    процедура р = проц (...) (...) ;
        ...
        локсправ := спрконст (
            х1 =// спр1 ,
            х2 = р) ;
        ...
        файл1 = файл (локсправ // х1 // фл1) ;
        прог1 := прогр (локсправ // х2) ;
        ...
конец
```

Здесь предполагается, что во внешнем контексте программы есть справочник *//спр1*, в который под именем "*фл1*" занесена ссылка на некоторый файл. Тогда с точки зрения выполнения второго оператора элемент "*х1*" эквивалентен элементу обычного справочника, содержащего ссылку на справочник *//спр1*. Изображенное ⟨внешнее имя⟩ приводит к файлу, который от-

крывается при выполнении этого оператора. В результате открытия программы в правой части третьего оператора выдается процедура *р*.

Пример 2. Использование константного справочника для формирования пакета программ (пакет тригонометрических функций).

```
программа
    % описание глобальных констант, текстов и пр.
  процедура синус = проц (арг) ... ,
      косинус =проц (арг) ... ,
      ...
      ;
    % результат открытия программы:
    спрконст (
          sin = синус ,
          cos = косинус ,
      ...
      )
конец
```

Результатом открытия этой программы является указатель, установленный на константный справочник. Если к файлу объектного кода этой программы приводит ⟨внешнее имя⟩ *//триг*, тогда обращение к какой-либо функции можно, например, сформировать следующим образом:

```
начало
    конст sin = прогр (// триг // sin) ;
    ф64
    ...
    z := sin (x);
    ...
конец
```

Косвенные элементы. Присутствие символа @ в ⟨элементе справочника⟩ означает, что содержимое элемента рассматривается как косвенная ссылка. При поиске по контексту подобные элементы играют такую же роль, как и элементы обычных справочников, содержащие косвенные ссылки.

15. Пакет заданий

Пакет заданий представляет собой колоду перфокарт, составленную из последовательности заданий. Задание — это пк-файл, имеющий структуру, изображенную на рис. 3.2.

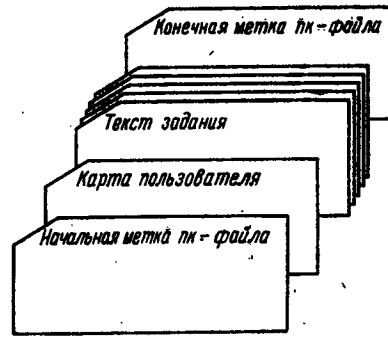


Рис. 3.2. Колода пакетного задания.

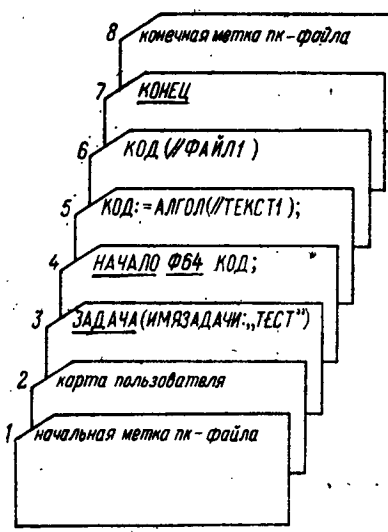


Рис. 3.3. Пример задания.

Карта пользователя — это перфокарта с именем пользователя, паролем и прочими атрибутами, позволяющими системе идентифицировать пользователя и найти его внешний контекст (см. Приложение 4). Текст задания — это конструкция ⟨закрытый оператор⟩ или ⟨запуск задачи⟩, в ⟨тексте программы⟩ которой описано содержание задания.

Выполнение пакета состоит в том, что задания вводятся одно за другим и в параллельном режиме инициируется их выполнение.

Выполнение задания состоит в выполнении конструкции ⟨запуск задачи⟩ или ⟨закрытого оператора⟩. Внешним контекстом, в котором происходит выполнение, является контекст, идентифицируемый картой пользователя. Пример задания представлен на рис. 3.3. На карте 3 задается имя задачи "*тест*". Это имя печатается при выдаче итогов решения задачи. На карте 5 задается трансляция алгол-транслятором текстового файла *//тест*1 (предполагается, что идентификатор *алгол* описан в стандартном контексте и обозначает процедуру, являющуюся алгол-транслятором). Указатель внешнего объекта, установленный на файл объектного кода, присваивается переменной *код*. На карте 6 файл объектного кода неявно открывается, и программа выполняется. В качестве фактического параметра передается файл *//файл*1.

ГЛАВА 4

АТРИБУТЫ ОБЪЕКТОВ И АТРИБУТЫ ЗАДАЧИ

В этой главе описываются основные атрибуты внешних объектов, оперативных объектов связи и буферов, атрибуты паспорта в-массива и атрибуты задачи. Описанию семантики атрибута предшествует следующая информация:

— идентификатор атрибута;

— тип внешнего устройства, на котором может располагаться внешний объект;

— принадлежность атрибута. Здесь используются следующие условные обозначения: О — атрибут открытия, В — атрибут внешнего объекта, ОО — атрибут, хранящийся в указателе объекта связи с файлом. В результате модификации атрибута последней разновидности создается новый экземпляр указателя того же оперативного объекта, причем в новом указателе содержится измененное значение атрибута;

— перечень конструкций, в которых можно использовать атрибут. Здесь используются следующие условные обозначения: Г — атрибут можно указывать в генераторе объекта (для атрибутов открытия — в ⟨генераторе объекта связи⟩, а для атрибутов внешнего объекта — в ⟨генераторе внешнего объекта⟩ или в ⟨генераторе объекта связи⟩ в случае неявной генерации внешнего объекта), М — возможна модификация атрибута, ГВ — для атрибутов открытия обозначает, что умолчание для этого атрибута можно хранить в заголовке внешнего объекта. Если использование атрибута в некоторой конструкции разрешено не для всех типов внешних устройств, то те типы, для которых оно разрешено, перечислены в скобках после обозначения данной конструкции;

— для атрибутов файла указано, сохраняется ли значение атрибута после открепления файла. Если значение не сохраняется, то это обозначается буквами НС;

— в квадратных скобках указан тип значения атрибута. Если атрибут можно задавать в генераторе, то следом указывается значение, которым атрибут инициализируется по умолчанию. Тип обозначается следующим образом: Б — истина или ложь, Ц — целое, С — указатель байтового вектора или набор, У — указатель объекта связи или с-вектора, М — процедура.

1. Общие атрибуты оперативных объектов

типоб

О; [Ц]

Значение атрибута определяет тип оперативного объекта. Атрибут может принимать следующие значения:

1 — заголовок открытого контейнера;
2 — заголовок открытого файла;
3 — позиционная переменная;
4 — таблица буферов;
5 — блок ввода/вывода;
6 — массив процедур реакций на ошибки обмена;
7 — аварийный объект связи;
8 — подвижный указатель внешнего объекта.

ошобмена

О/М/Г; [У,пусто32]

Если значением атрибута является массив процедур реакции (МПР), то для всех операций, осуществляемых с помощью данного объекта связи, имеют силу указанные в МПР переопределения реакций на ошибки обмена.

локал

О; [Б, истина]

Если значение атрибута истина, то данный оперативный объект — локальный, а иначе — глобальный.

2. Атрибуты файла

2.1. Характеристики внешнего устройства.

типву

любой; В/Г; [Ц]

Значение атрибута определяет тип внешнего устройства. Возможны следующие значения:

1 — магнитный барабан (мб);
2 — магнитный диск (мд);
3 — магнитная лента (мл);

4 — ввод с перфокарт (чпк);
5 — вывод на перфокарты (зпк);
6 — ввод с перфоленты (чпл);
7 — вывод на перфоленту (зпл);
8 — алфавитно-цифровой дисплей (ацд);
9 — пишущая машинка (пм);
10 — алфавитно-цифровое печатающее устройство (ацпу).

плотность

мл; В/Г; [Ц, 0]

Атрибут определяет плотность записи на магнитную ленту. Возможны следующие целочисленные значения:

0 — 8 байтов/мм;
1 — резерв;
2 — 32 байта/мм;
3 — 64 байта/мм.

кодблока

чпл, зпл; О/Г; [Б, ложь]

Если значение атрибута — истина, то конец блока на перфоленте обозначается специальным символом, набираемым на панели устройства.

двоичный

зпк, чпк; В/Г; [Б, ложь]

Если значение атрибута — истина, то информация на перфокартах представлена в двоичном виде, в противном случае — в перфокарточном коде КПК-12.

2.2. Характеристики физических границ.

максдлинблока

любой; В(мб,мд,мл,пк,ацпу)/О(пл,ацд,пм)/Г/М; [Ц, см. семантику]

Значение атрибута определяет максимальное число элементов в блоке файла и тем самым задает размер буфера, используемого при буферизованном обмене. Для мб- и мд-файлов число элементов задается в словах, для файлов остальных типов — в байтах. Умолчание выбирается в зависимости от типа: для мб-, мд-файлов — 256 слов; для мл-, пл-файлов — 256*8 байтов; для пк-файлов, если значение атрибута *двоичный* есть истина, то 160 байтов, в противном случае — 80 байтов; для ацпу файлов — 128 байтов; для пм-, ацп-файлов — 80 байтов.

Атрибут можно модифицировать только тогда, когда файл не открыт для буферизованного обмена.

Соответствие значений атрибутов *максдлинблока* и *длинлиста* мб- и мд-файлов проверяется при явном указании этого

атрибута в генераторе файла, при модификации этого атрибута, при открытии файла для буферизованного обмена (однобуферного или многобуферного). В последнем случае может быть явно указано новое значение атрибута *максдлинблока*. Если оно не указано, то при проверке используется текущее значение атрибута. Проверка состоит в следующем. Если в текущий момент для файла еще не отведена память (значение атрибута *чслфлистов* равно 0), то значение атрибута *длинлиста* корректируется таким образом, чтобы стать ближайшим кратным значению атрибута *максдлинблока*. Если же память уже отведена и значение *длинлиста* не кратно значению *максдлинблока*, то корректируется *максдлинблока*.

длинлиста
мб, мд; В/Г/М; [Ц, 128]

Значение атрибута определяет количество секторов в физическом листе памяти мб- и мд-файла. После того как для файла отведен первый физический лист, атрибут можно только запрашивать.

максдлинфайла
любой; О/Г/М/ГВ(мб,мд); [Ц, см. семантику]

Значение атрибута определяет максимально возможный объем файла. Объем мб- и мд-файлов задается в терминах числа листов, а объем других файлов — в терминах числа блоков. Умолчание выбирается в зависимости от типа: мб-, мд-файл — 10 листов, мл-, пк-, пл-, ацпу-файл — $(10^6 - 1)$ блоков.

чслфлистов
мб, мд; В/Г; [Ц, 0]

Значение атрибута определяет число физических листов, отведенных в данный момент для файла. Начальное значение атрибута задает число листов, отводимых для файла в момент его создания.

максфлист
мб, мд; В; [Ц]

Значение атрибута определяет максимальный номер математического листа, для которого существует соответствующий ему физический лист.

обрезан
мб, мд; В/Г/М; [Ц, 0]

Атрибут может принимать следующие значения: 0 — при откреплении файла система не будет осуществлять сокращение

последнего листа файла; 1 — при откреплении файла система произведет сокращение последнего листа в соответствии с атрибутом *блоклкф* и освободившийся остаток отдаст в область свободной памяти на носителе; 2 — система произвела сокращение последнего листа (объем такого файла в дальнейшем нельзя увеличивать).

максдлинстроч

апцу; О/Г/ГВ; [Ц, 128]

Значение атрибута определяет максимальное количество литер на строчке бумажного носителя апцу-файла.

максдлинстран

апцу; О/Г/М/ГВ; [Ц, 0]

Если значение атрибута отлично от нуля, то оно определяет максимальное количество строчек, располагающихся на логической странице бумажного носителя апцу-файла.

2.3. Характеристики логических границ.

блоклкф

мб, мд; В/М; [Ц]

Значение атрибута определяет номер первого свободного блока файла.

элемлкф

мб, мд; В/М; [Ц, 0]

Значение атрибута задает позицию логического конца файла в последнем заполненном блоке.

длинблока

мл, чпл, ацд, пм; О/М; [Ц]

Значение атрибута определяет реальную длину текущего блока файла (в байтах).

2.4. Характеристики текущей позиции.

блокфайла

мл, пк, апцу, пл; О/М; [Ц]

Значение атрибута определяет порядковый номер текущего блока (относительно начала файла), в конец которого реально установлена головка записи/считывания. Атрибут может принимать значения типа набор: "*начф*", если текущая позиция установлена на начало файла, и "*конф*", если текущая позиция установлена на конец файла.

строчка

апцу; О/Г/НС; [Ц]

Если значение атрибута *максдлинстран* отлично от нуля, то значение атрибута *строчка* определяет номер текущей строчки. Номер строчки отсчитывается от начала текущей логической страницы (номер первой строчки равен 1). При значении атрибута *максдлинстран*, отличном от нуля, начальное значение атрибута *строчка* равно 1, в противном случае — 0.

страница
ацпу; О/Г/М; [Ц]

Если значение атрибута *максдлинстран* отлично от нуля, то значение атрибута *страница* определяет номер текущей логической страницы.

При значении атрибута *максдлинстран*, отличном от нуля, начальное значение этого атрибута равно 1, в противном случае — 0. Номер текущей страницы увеличивается на 1, когда номер строчки становится равным значению атрибута *максдлинстран*.

2.5. Атрибуты, связанные с именованием.

имяфайла
мл, пк, ацпу, пл; В(мл,пк,ацпу)/Г(мл,пк,ацпу);
[С, пустое имя(пк,пл,ацпу)]

Значением атрибута является указатель байтовой строки или байтовый набор. В обоих случаях последовательность символов — это имя из метки файла.

Строка нулевой длины или пустой набор задают «пустое имя». Файлы с такими именами называются непомеченными. Непомеченный файл не имеет метки, где хранились бы его атрибуты. Поэтому атрибуты непомеченного файла являются атрибутами открытия. Стратегия постановки на устройство нужного непомеченного файла определяется оператором. Пл-файл всегда является непомеченным.

Умолчание для мл-файлов состоит в следующем. Если при создании файла атрибут *имяфайла* не был указан, то файл не прикрепляется ссылкой к внешнему контексту (т. е. является временным). Результат последующего запроса атрибута *имяфайла* не определен.

2.6. Даты.

датасозд
мд, мб, мл; В; [С]

Значение атрибута определяет дату создания объекта в формате *ггпддд*, где *гг* — две последние цифры года, *п* — пробел, а *ддд* — номер дня в году.

даталиквид
мд, мб, мл; В/Г/М(мб, мд); [С, дата создания + 7]

Значение атрибута определяет дату ликвидации файла (в формате *ггпддд*). Если программа открывает файл с истекшим сроком хранения, то оператору посылается сообщение об этом.

послдата
мб, мд; В; [С]

Значение атрибута определяет дату последнего использования файла (в формате *ггпддд*).

2.7. Характеристики распределения листов.

файлвпакет
мд; В/Г; [Б, ложь]

Если значение атрибута — **истина**, то физические листы данного файла располагаются на одном пакете дисков.

листвцилиндр
мд; В/Г [Б, ложь]

Если значение атрибута — **истина**, то каждый физический лист файла полностью располагается на одном цилиндре.

порядоклистов
мд; В/Г; [Ц, 0]

Если значение атрибута равно числу 1, то физические листы файла располагаются на диске в порядке возрастания номеров соответствующих математических листов. Если значение атрибута равно числу 2, то листы располагаются в порядке убывания. Если значение атрибута равно нулю, то листы располагаются в произвольном порядке.

упп
мд, мб; В/Г; [Б, ложь]

Атрибут задает режим управляемого программистом распределения памяти на носителе. Это означает, про программист с помощью атрибута *базалиста* должен указывать на начальные адреса каждого листа файла.

Режим управляемого распределения памяти для мб-файлов можно задавать только в привилегированном режиме.

2.8. Характеристики текстовых файлов.

длинстрок
любой; В/Г; [Ц, 0]

Если значение атрибута отлично от нуля, то файл состоит из строк одинаковой длины, равной этому значению. В противном случае файл состоит из строк переменной длины. В обоих случаях длина строк не должна превышать значения атрибута *максдлинблока*.

пакстрок
любой; В/Г(мб, мд, мл); [Б, **истина**(мб, мд, мл)/**ложь** (пк, пл, ацпу,ацд,пм)]

Если значение атрибута — **истина**, то пробелы внутри строк упаковываются программой, создающей такой текстовый файл. Это означает, что в состав строк могут входить байты, кодирующие некоторое число подряд расположенных пробелов. Числовая величина таких байтов заключена в пределах $1 \leqslant \kappa \leqslant 63$, где κ — количество представленных таким способом пробелов.

Если значение этого атрибута — **ложь**, то упаковка пробелов не производится.

длинфнс
любой; В/Г/М(ацд, пм); [Ц, 0]

Если значение атрибута отлично от нуля, то файл является нумерованным, т. е. каждая строка содержит собственный номер, состоящий из *длинфнс* цифр. В противном случае в файле плавающая нумерация строк, т. е. строка не содержит своего номера.

кодфнс
любой; В/Г(мб, мд, мл); [Ц, 0(мб, мд, мл)/1(пк, пл, ацпу, ацд, пм)]

Для нумерованного файла значение этого атрибута определяет кодировку номера строки. Если значение равно нулю, то кодировка двоичная. В этом случае для номера строки отводится столько байтов, сколько нужно, чтобы уместить двоичный код максимального целого, состоящего из *длинфнс* цифр. Например, при значении *длинфнс*, равном числу 8, максимальное целое занимает 27 битов, и для номера строки отводится 4 байта. Если значение атрибута равно 1, используется символьная кодировка. В этом случае для номера строки отводится *длинфнс* байтов, и номер кодируется в виде последовательности литер, соответствующих цифрам номера.

позфнс
любой; В/Г(зпк,ацпу)/М(ацд); [Ц, 0]

Для нумерованного файла значение этого атрибута определяет номер позиции в строке, начиная с которой располагается номер строки.

фнс
любой; О/М; [Ц, 0]

Для нумерованного файла значение этого атрибута равно фиксированному номеру текущей строки.

шагфнс
любой; О/М; [Ц, 1]

Для нумерованного файла значение этого атрибута задает шаг нумерации строк.

кодпрогр
мб,мд; В/М; [Ц, пусто32]

Значением атрибута является целое число, определяющее номер того элемента СВС (справочника внешних связей) текстового файла, который содержит ссылку на файл объектного кода, полученного при трансляции данного текста.

При модификации этого атрибута значением типа указатель внешнего объекта ссылка, содержащаяся в указателе, помещается не в атрибут, а в соответствующий элемент СВС.

имяяз
мб, мд; В/М; [С, пустая строка]

Содержимое атрибута — это строка литер, задающая имя программы транслятора языка программирования, на котором написан данный текст. Имя ищется в контексте данного файла (см. атрибут *внешконт*). Если значение атрибута — пустой набор или пустая строка, то содержимое файла — это произвольный текст.

2.9. Прочие характеристики файла.

типфайла
любой; В/Г/М; [Ц, 0]

Значение атрибута характеризует структуру информации, составляющей содержимое файла:

0 — данные произвольной структуры;
1 — файл, содержащий псевдофайл;
2 — файл объектного кода;
3 — текст.

Псевдофайл предназначен для имитации файлов с одним типом внешнего устройства на носителях внешних устройств другого типа.

областьвав
мб, мд; В/Г; [Ц/У, 0]

Значение атрибута, указанное при создании файла, определяет размер области пользователя в заголовке файла. В этой области пользователь может хранить свою информацию. При запросе этого атрибута выдается указатель, описывающий эту область как с-вектор элементов формата ф64.

длинобластьбуф
любой; В/Г/М; [Ц, 0]

Значение этого атрибута определяет длину области пользователя в буфере. Атрибут можно модифицировать только тогда, когда файл не открыт для буферизованного обмена. Например, это можно делать, если файл находится в закрытом состоянии.

типданных
мб, мд, мл, пк; ОО/Г/ГВ/М; [Б, **ложь**]

В зависимости от значения атрибута *типданных* информация о типе данных теряется при обмене или сохраняется. Если значение атрибута есть **ложь**, то при записи теги отбрасываются. При чтении приписывается тип полный набор. Если значение атрибута есть **истина**, то запись и считывание не приводят к потери информации о типе. В этом случае вводятся или выводятся все 72 разряда каждого участвующего в обмене слова оперативной памяти (т. е. 9 байтов). Из них 64 разряда информационных, а остальные 8 разрядов занимает информация о типе и четности.

Для мд- и пк-файлов имеются следующие особенности. Сектор мд-носителя логически состоит из 28 72-разрядных элементов, т. е. в нем упаковывается 28 полных слов. На перфокарте пк-файла упаковывается 8 полных слов, если значение атрибута *двоичный* есть **ложь**, и 17 полных слов, если значение этого атрибута **истина** (см. семантику атрибута *двоичный*).

Обмен данными типа адресной информации в непривилегированном режиме приводит к возникновению ошибки «привилегированный обмен».

Модифицировать этот атрибут можно только в процессе непосредственного обмена.

Файлы основных типов имеют следующее значение этого атрибута: мд-файл объектного кода — **ложь**; мб-файл объектного кода — **истина**; текстовый файл — **ложь**.

восстановимый
мб, мд; В/Г; [Б, **ложь**]

Если значение этого атрибута — истина, то система с целью более точного восстановления при сбое регулярно прописывает ряд блоков, находящихся за пределами текущего логического конца файла, информацией, имеющей стандартный «рисунок».

максоблпоиска
мб, мд; В/Г; [Ц, 9999]

Атрибут определяет, через какое количество блоков, записанных в конец файла, необходимо корректировать копию заголовка на внешнем носителе. Обычно этот атрибут используется вместе с атрибутом *восстановимый*. Чем меньше значение этого атрибута, тем выше частота копирования заголовка, и тем меньше область поиска конца файла при сбое.

облпоиска
мб, мд; В/Г; [Ц, 9999]

Значение атрибута показывает, на сколько блоков увеличился файл со времени последнего копирования заголовка на внешний носитель. Может принимать значения от нуля до *максоблпоиска* — 1. Присваивание атрибуту нулевого значения приводит к копированию заголовка файла на внешний носитель.

наччслбуферов
любой; О/Г/М/ГВ; [Ц, 2]

Значение атрибута определяет, какое число буферов создается для файла в момент его первого открытия для буферизованного обмена в рамках каждой задачи.

Атрибут можно модифицировать только тогда, когда файл не открыт для буферизованного обмена.

запасбуферов
любой; О/Г/ГВ/М; [Ц, 0]

Значение атрибута определяет число резервных буферов на каждое открытие данного файла.

Атрибут можно модифицировать только тогда, когда файл не открыт для буферизованного обмена.

приглашение
адд, пм; О/Г/М; [Ц, 0]

Значение атрибута задает режим диалога, осуществляемого с помощью терминального файла. Нулевое значение атрибута задает режим «без приглашения». В этом режиме операция чтения работает обычным образом. Значение атрибута, равное 1, задает режим «с приглашением в виде номера строки»,

В этом режиме операция чтения сначала осуществляет приращение текущего значения атрибута *фнс* на значение атрибута *шагфнс* и выводит на носитель номер новой строки в виде *длинфнс* литер. Затем осуществляет чтение ответного сообщения. Если производилась модификация атрибута *фнс*, то первая из последующих операций чтения не осуществляет приращение вновь установленного значения.

Если значение этого атрибута больше 1, осуществляется режим «с приглашением в виде символа». В этом режиме приглашение выводится в виде литеры, числовое значение которой равно значению этого атрибута.

очрдввода
ацд, пм; О; [Ц]

Значение атрибута определяет количество сообщений в очереди входных сообщений.

активность
ацд, пм; О; [М]

Если значением атрибута является метка процедуры, то эта процедура запускается при возникновении независимой активности со стороны терминала. Под независимой активностью понимается факт поступления с терминала сообщения, не затребованного программой (сообщение может быть затребовано программой с помощью операции чтения).

сообщение
мб, мд, мл, зпк; зпл, ацпу; О/Г/ГВ(мб, мд)/М; [С; пусто32]

Значением атрибута является указатель, описывающий байтовый вектор, который содержит сообщение, посылаемое оператору в момент открытия файла.

времяобмена
любой; О; [Ц]

Значение атрибута определяет суммарное время, затраченное на обмен с данным файлом.

смфдоступа
любой; О; [У]

Значением атрибута является указатель переменной, находящейся в заголовке файла и содержащей семафор. С помощью этого семафора пользователи, одновременно работающие с одним и тем же файлом, могут синхронизироваться по доступу к файлу. Следует учитывать, что система не выполняет над этим семафором никаких неявных операций.

длинсвс

мб, мд; В/Г/М; [Ц, 0]

Значение атрибута задает количество элементов в справочнике внешних связей файла. Значение атрибута можно изменять в сторону увеличения.

внешконт

мб, мд; В/М; [Ц, пусто32]

Значением атрибута является целое число, определяющее номер элемента в справочнике внешних связей файла объектного кода, содержащего ссылку на внешний контекст программы.

При модификации этого атрибута значением типа указатель внешнего объекта ссылка, содержащаяся в указателе, помещается в соответствующий элемент справочника внешних связей.

Запись целого обычно делает транслятор, а пользователь затем записывает ссылку на внешний контекст.

В случае, если текст (или файл с данными произвольной структуры) содержит внешние имена файлов, к которым необходимо обращаться в процессе чтения текста, этот же атрибут можно использовать и для ссылки из текстового файла (см. также атрибут *имяяз* текстового файла).

текстпрогр

мб, мд; В/М; [Ц, пусто32]

Атрибут определяет номер того элемента справочника внешних связей файла объектного кода, который содержит ссылку на файл исходного текста программы. В остальном семантика аналогична атрибуту *внешконт*.

инфпровр

мб мд; В/М; [Ц, пусто32]

Атрибут определяет номер того элемента справочника внешних связей файла объектного кода (ФОК), который содержит ссылку на расширение ФОК'а (локальные словари программы, справочник локальных словарей и прочая информация, используемая системой при выдаче сообщений об аварийном прекращении выполнения программы, для символьной отладки программ, при комплексации независимо транслированных программ и т. д.). В остальном семантика аналогична атрибуту *внешконт*.

имяпол

мб, мд, мл; В; [С]

Значением атрибута является указатель байтовой строки, содержащей имя пользователя, создавшего данный файл.

вакрзап
мб, мд, мл; ОО/М; [Б, ложь]

Если значение атрибута — **истина**, то файл можно использовать только для чтения

переполнен
мл, чпл, апд, пм; О; [Ц]

Если после считывания очередного блока файла оказывается, что его реальная длина больше, чем длина массива, в который производится считывание, то значением этого атрибута становится 1, если длины равны, то значением атрибута становится 0, в противном случае значением атрибута становится —1.

вапконтроль
мб, мд; О/М; [Б, ложь]

Если значение атрибута — **истина**, то запись на носитель осуществляется с последующим контрольным считыванием.

3. Атрибуты листа

класслиста
мб, мд; В [Ц, 0]

Атрибут *класслиста* определяет барабан (мд-пакет), на котором размещается данный лист.

Если отводится участок памяти для листа нулевого класса, то занимается память на любом барабане (мд-пакете), на котором есть свободный участок нужной длины. Для листа ненулевого класса отводится память на барабане, класс которого равен классу листа. Если таких барабанов нет (или есть, но на них нет свободной памяти нужной длины), то отводится память на каком-либо барабане (мд-пакете) нулевого класса, на котором есть свободная память нужной длины. Этому барабану приписывается класс листа. Барабану возвращается нулевой класс после того, как освобождаются все его участки, отданные под листы с ненулевым классом. Таким образом, данный атрибут позволяет группировать физические листы файла на одном устройстве или на группе устройств с одинаковым классом. Класс должен быть приписан математическому листу до того, как распределяется память для соответствующего физического

листа. После того как память для физического листа отведена, изменять класс данного листа нельзя.

распределен
мб, мд; В/М [Б, ложь]

Если значение атрибута — **истина**, то это означает, что для данного математического листа распределен соответствующий физический лист. Отведение физического листа, соответствующего некоторому математическому, осуществляется либо неявно (в момент первой записи в этот лист), либо явным образом, с помощью модификации атрибута *распределен*. Если значением атрибута *распределен* становится **ложь**, то освобождается участок, занятый соответствующим физическим листом. Освободившийся участок поступает в область свободной памяти на носителе. Если файл используется в буферизованном обмене, то ликвидируются все буфера, соответствующие блокам, расположенным на освобождаемом листе.

При попытке распределения листа, находящегося за пределами максимально возможного объема файла, возникает ошибка «физический конец файла».

базалиста
мб, мд; В/М; [Ц]

Значение атрибута определяет адрес начала листа на носителе (номер математического пакета плюс физический адрес начала листа внутри пакета). В непривилегированном режиме можно только запрашивать значение этого атрибута. В привилегированном режиме, при значении атрибута *упп* — **истина**, можно устанавливать и изменять значение этого атрибута.

4. Атрибуты контейнера

4.1. Характеристики внешнего устройства.

типву
мд, мл; В; [Ц]

Значение атрибута определяет тип внешнего устройства, на котором располагается контейнер. Атрибут может иметь следующие значения: 2 — магнитный диск; 3 — магнитная лента.

4.2. Характеристики физических границ.

максдлинконт
мд, мл; В/Г/ (мд); [Ц]

Значение атрибута определяет максимально возможный объем контейнера в терминах количества томов.

чслфпак
мд; В/Г; [Ц, 1]

Значение атрибута определяет число физических томов, отведенных для мд-контейнера в данный момент. Начальное значение этого атрибута задает число томов, отводимых для контейнера в момент его создания.

максфпак
мд; В; [Ц]

Значение атрибута определяет максимальный номер математического тома, для которого существует соответствующий ему физический том.

4.3. Характеристики текущей позиции.

томконт
мл; О/М; [Ц]

Значение атрибута определяет номер текущего тома относительно начала контейнера. Кроме целочисленных значений, атрибут может принимать значения типа набор: *"нач"* — начало контейнера, т. е. начало первого тома контейнера; *"конк"* — конец контейнера, т. е. конец последнего тома контейнера.

файлконт
мл; О/М; [Ц]

Значение атрибута определяет порядковый номер текущего файла относительно начала контейнера. При модификации атрибута текущая позиция перемещается на файл с заданным номером.

нмртома
мд, мл; О/Г/М; [Ц]

Для мл-контейнера значение этого атрибута определяет регистрационный номер текущего тома. При модификации атрибута текущая позиция перемещается на том с заданным регистрационным номером.

Указание этого атрибута в генераторе мд-контейнера (мл-контейнера) означает, что для базового пакета (начальной ленты) используется том с заданным регистрационным номером.

позтома
мл; О/М; [С]

Атрибут используется для уточнения текущей позиции. Установка позиции происходит при модификации атрибута. Атри-

бут может принимать следующие значения типа набор: "*начф*" — начало текущего файла; "*конф*" — конец текущего файла; "*начт*" — начало текущего тома; "*конт*" — конец текущего тома.

направление
мл; О/М [Б, истина]

Атрибут определяет позицию, в которую перематываются файлы контейнера при закрытии. При прямом направлении (истина) файл сматывается на конец, а при обратном (ложь) — на начало.

смонтирован
мд, мл; В/М; [Б, ложь]

Если значение атрибута истина, то по крайней мере один из томов смонтирован на устройство. Атрибут можно запрашивать или присваивать ему значение ложь.

4.4. Прочие атрибуты.

имяконт
мд, мл; В/Г/М/(мд); [С]

Семантика атрибута аналогична семантике соответствующего атрибута файла (*имяфайла*)

имяпол
мд, мл; В; [С]

Значением атрибута является указатель байтовой строки, содержащей имя пользователя, создавшего данный контейнер.

5. Атрибуты тома

распределен
мд; В/М; [Б, ложь]

Если значение атрибута истина, то для данного математического тома распределен некоторый физический том. Отведение физических томов происходит либо неявно, в момент переполнения уже распределенных томов контейнера, либо явным образом, с помощью модификации атрибута *распределен*.

нмртома
мл, мд; В/Г; [Ц]

При модификации атрибута для математического тома распределяется физический том с заданным регистрационным номером.

упп
мд; В/Г; [Б, ложь]

Если значение атрибута **истина**, то для данного пакета устанавливается режим управляемого программистом распределения памяти. Это означает, что внутри данного пакета можно распределять память для файлов с управляемым распределением памяти (см. аналогичный атрибут для файла). Устанавливать значение этого атрибута можно только до того, как пакет начал использоваться для создания в нем файлов.

смонтирован
мд, мл; В/М; [Б, ложь]

Если значение атрибута **истина**, то том контейнера с данным математическим номером смонтирован на устройство. Атрибут можно запрашивать или присваивать ему значение ложь.

6. Атрибуты буфера

блокфайла
любой; О; [Ц]

Значение атрибута определяет номер блока, которому соответствует буфер. Кроме целочисленных значений, атрибут может принимать значения типа набор: "*начф*" — позиция «начало файла»; "*конф*" — позиция «конец файла». Следует учитывать, что в этих случаях реальный буферный массив не создается.

длинблока
любой; О/М; [Ц]

Значение атрибута определяет длину блока, считанного в буфер (при чтении), или длину части буферного массива, записываемой в файл.

модифицирован
любой; О/М; [Б]

Значение атрибута определяет, запрашивался ли указатель для записи в буферный массив.

переполнен
любой; О; [Ц]

Если в результате считывания блока в буфер оказывается что реальная длина блока больше, чем значение атрибута *максдлинблока*, то значением этого атрибута становится 1, если дли-

ны равны, то значением атрибута становится 0, в противном случае значением атрибута становится —1.

областьбуф
любой; О; [Ц]

Значением атрибута является указатель, описывающий область пользователя в буфере.

строчка, страница
ацпу; О; [Ц]

Семантика атрибутов *строчка*, *страница* аналогична семантике соответствующих атрибутов файла с той разницей, что рассматривается не реальная текущая позиция, а логическая, т. е. та, которая задается атрибутом *блокфайла* текущего буфера.

смфобмена
любой; О; [У]

Значением атрибута является указатель семафора завершения операции обмена с данным буфером.

7. Атрибуты позиционной переменной

С помощью позиционной переменной можно запрашивать все атрибуты текущего буфера. Кроме того, существует ряд атрибутов открытия, хранящихся непосредственно в позиционной переменной. Эти атрибуты перечислены ниже.

формэлем
любой; О/Г/М/ГВ(мб, мд); [Ц, 6(мб, мд)/3(мл, пк, ацпу, пл, ацд)]

Значение атрибута определяет формат элемента буферного массива. Атрибут может принимать следующие значения: 0 — ф1; 2 — ф4; 3 — ф8; 5 — ф32; 6 — ф64; 7 — ф128.

элемблока
любой; О/М; [Ц]

Значение атрибута определяет номер текущего элемента блока.

прикрепфайл
любой; О; М [Ц]

Значением атрибута является заголовок открытого файла, прикрепленного к данной позиционной переменной.

дисклента
мб, мд, мл; В/Г; [Б, **ложь**]

Мб- и мд-файлы отличаются от мл-файлов как с точки зрения семантики продвижения позиции логического конца файла, так и с точки зрения длины блоков памяти файла. Если значение атрибута *дисклента* — **истина**, то это различие уничтожается, т. е. программа, написанная для мб- или мд-файла, будет приводить к тем же результатам при использовании ее для мл-файла и наоборот. При работе с таким файлом можно пользоваться только теми операциями, которые разрешены для мл-файлов. Из разрешенных операций нужно дополнительно исключить модификацию атрибута *длинблока*. В результате память файла будет состоять из блоков длиной *максдлинблока*, а размер заполненной части файла будет устанавливаться, как и в случае обмена с мл-файлами.

областьпозп
любой; О/Г; [Ц/У, 0]

Значение атрибута, указанное при создании позиционной переменной, определяет в словах размер области пользователя в позиционной переменной. В этой области пользователь может хранить свою информацию. При запросе этого атрибута выдается указатель, описывающий область.

8. Атрибуты блока ввода / вывода

смфобмена
любой; О/М; [У]

Значением атрибута является указатель семафора завершения параллельного обмена с использованием данного БВВ.

вобмене
любой; О; [Б]

Если значение атрибута — **истина**, то операция обмена с участием данного БВВ еще не окончена.

прервцп
любой; О/Г/М; [Б, **ложь**]

Если значение атрибута — **истина**, то по завершении очередного обмена происходит прерывание центрального процессора. Атрибут можно задавать только в привилегированном режиме.

опобмена
любой; О; [Ц]

Значение атрибута определяет код аварийной операции обмена. Данный атрибут и атрибуты *адробмена* и *масобмена* принадлежат аварийному БВВ (см. гл. 5).

адробмена
мб, мд; О; [Ц]

Значение атрибута определяет адрес начала аварийного обмена.

масобмена
любой; О; [У]

Значением атрибута является указатель массива, с которым осуществлялась аварийная операция обмена.

9. Атрибуты паспорта

Ниже приводятся атрибуты объекта типа паспорт. Значение атрибута *нигрнуль* можно запрашивать и изменять. Остальные атрибуты можно только запрашивать. Атрибуты *длинизм* и *нигрнуль* определены только для прямоугольного параллелепипеда (см. атрибут *формпар*).

чслизм

Значение атрибута равно числу измерений выстроенного массива.

длинизм

С помощью этого атрибута можно запрашивать длины по измерениям. Для этого в конструкции ⟨запрос атрибута⟩ на месте компоненты ⟨уточнение⟩ надо указать либо номер измерения, например *читатр* (*а*, 1, *длинизм*), либо с-вектор, длина которого равна числу измерений, например

читатр (*а*, *вд*, *длинизм*).

В первом случае значением конструкции является длина заданного измерения, а во втором элементам вектора присваиваются длины всех измерений; значением конструкции является указатель вектора *вд*.

нигрнуль

Если значение атрибута — истина, то нижние границы по всем измерениям равны нулю. В противном случае значение ат-

рибута — **ложь**. С помощью ⟨модификации атрибутов⟩ этому атрибуту можно придавать значение **истина**. После такой установки паспорт будет по-прежнему описывать исходный массив, но с учетом сдвига всех нижних границ в нуль.

формпар

Если массив имеет форму прямоугольного параллелепипеда, то значение этого атрибута — **истина**; в противном случае значение — **ложь**.

базвект

Значением атрибута является указатель базового вектора, т. е. с-вектора, в котором расположен в-массив.

10. Атрибуты задачи

пам
[Ц]

Текущее значение объема оперативной памяти, используемой задачей.

резидпам
[Ц]

Текущее значение объема резидентной оперативной памяти, используемой задачей.

времяцп
[Ц]

Текущее значение времени, в течение которого задача занимала центральный процессор.

времяобмена
[Ц]

Текущее значение времени, потраченного на обмен.

размацпу
[Ц]

Текущее значение объема всех выдач из задачи на ацпу.

максматпам
Г; [Ц]

Максимальный объем математической памяти, которую может использовать задача. Этот атрибут и все другие, определяющие предельные значения объема ресурсов, задавать не обя-

зательно. Они предназначены для того, чтобы ограничить возможность неконтролируемых ошибок, приводящих к тому, что программа начинает неограниченно потреблять ресурс, в данном случае — математическую память.

максвремяцп
Г; [Ц]

Максимальное время, в течение которого задача может занимать центральный процессор.

максвремяобмена
Г; [Ц]

Максимальное время, которое может быть затрачено задачей на обмен.

максразмацпу
Г; [Ц]

Максимальный объем всех задач на ацпу.

статус
[Ц]

Текущее состояние задачи: 0 — во входной очереди; 1 — взята на решение; 2 — находится во временно приостановленном состоянии; 3 — печатаются виртуальные файлы вывода.

имязадачи
Г; [С]

Значением атрибута является указатель, описывающий литерный вектор, или набор. В обоих случаях это идентификация задачи, которая печатается при выводе итогов работы данной задачи.

генбарабан, гендиск, генмдконт, генмлконт, гензпк, гензпл, генацпу
Г; [М, стандартная процедура]

Значением каждого из этих атрибутов является процедура создания внешнего объекта. По умолчанию — это стандартные процедуры создания соответственно мб-файла, мд-файла на стандартном системном мд-контейнере, мд-контейнера, мл-контейнера, зпк-файла, зпл-файла, ацпу-файла. Этим атрибутам соответствуют стандартные идентификаторы *барабан, диск, мдконт, млконт, зпк, зпл. ацпу.*

чслпарпроц
[Ц]

Значением атрибута является число параллельных процессов, запущенных из задачи в данный момент.

жрнлвадачи
[У]

Значением атрибута является заголовок файла, содержащего «журнал задачи» (сообщения системы о ходе решения задачи).

имяпол
[С]

Значением атрибута является указатель литерной строки, содержащей имя пользователя, создавшего данную задачу.

виртвыв
[У]

Если в задаче используются строго выводные файлы (ацпу, энк, энл), то в процессе выполнения задачи такие файлы моделируются на быстрых носителях (мб, мд). Эти модели называются файлами виртуального вывода. По окончании задачи эти файлы выводятся на соответствующие устройства.

Для файлов виртуального вывода создается справочник. Ссылка на файл заносится в элемент справочника, имя которого совпадает с именем файла (атрибут *имяфайла*). Значением атрибута *виртвыв* является указатель внешнего объекта, установленный на справочник файлов виртуального вывода задачи.

файлработы
[У]

Значением атрибута является файл, содержащий объектный код, полученный в результате трансляции ⟨текста программы⟩ данной задачи (см. п. 11.3 гл. 2).

приорвадачи
Г; [Ц]

Значением атрибута является число, определяющее приоритет выполнения задачи.

нмрзадачи
[Ц]

Значением атрибута является номер задачи.

максвремя
Г; [Ц]

Значением атрибута является максимально возможное время решения данной задачи.

нмрехочрд
[Ц]

Значением атрибута является номер входной очереди, через которую задача попала на выполнение.

режимзадачи
Г; [Ц]

Значение атрибута определяет режим выполнения задачи: 0 — пакетный режим, 1 — диалоговый.

ГЛАВА 5

ОШИБКИ ВЗАИМОДЕЙСТВИЯ С ВНЕШНИМИ ОБЪЕКТАМИ

В процессе обмена и при работе с архивом могут возникнуть особые обстоятельства, объединяемые под общим названием «ошибки взаимодействия с внешними объектами». Для каждой подобной ошибки система предусматривает стандартную реакцию. Пользователь имеет возможность переопределить стандартную реакцию, т. е. указать действие, которое необходимо выполнить при возникновении ошибки. Часть ошибок имеет аварийный характер, и стандартная реакция на них состоит в том, что выполнение системной процедуры, реализующей операцию над внешним объектом, прекращается, и выполняется структурный переход по динамической ситуации, соответствующей обнаруженной ошибке. Если задача не определила реакцию на данную ситуацию, то выполнение задачи будет прекращено. Стандартная реакция на другую часть ошибок состоит в том, что системная процедура предпринимает некоторые действия, не приводящие к прекращению ее выполнения. В результате этого работа программы может быть нормально продолжена.

1. Номенклатура ошибок

Ниже перечисляются ошибки взаимодействия с внешними объектами и описываются стандартные реакции на каждую ошибку.

1) Ошибка «логический конец файла» возникает в следующих случаях:

— при попытке чтения информации, находящейся за пределами заполненной части файла (это не распространяется на чтение информации из мб-, мд-файла при непосредственном обмене, так как в этом случае система не отслеживает позицию логического конца файла);

— при попытке установить текущую позицию (с помощью операции позиционирования при непосредственном обмене) за пределы заполненной части файла.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситлкф*.

2) Ошибка «логический конец контейнера» возникает при попытке установить текущую позицию за пределы контейнера.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситлкк*.

3) Ошибка «физический конец файла» возникает при попытке увеличения объема файла до размера, превышающего максимально возможный (см. атрибут *максдлинфайла*).

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситфкф*.

4) Ошибка «физический конец контейнера» возникает при попытке увеличения объема контейнера до размера, превышающего максимально возможный (см. атрибут *максдлинконт*).

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситфкк*.

5) Ошибка «конец тома» возникает в следующих случаях:

— при чтении блока мл-файла, находящегося вне текущего тома данного мл-файла;

— при записи блока мл-файла, не умещающегося на текущий том данного мл-файла;

— при установке текущей позиции на блок, находящийся вне текущего тома данного мл-файла;

— при создании нового файла, начальная метка которого не умещается на текущий том данного мл-контейнера;

— при установке текущей позиции на файл, находящийся вне текущего тома данного мл-контейнера.

Стандартная реакция: происходит переключение на нужный том мл-контейнера (при генерации нового файла или записи нового блока назначается следующий том), и этот том становится текущим.

6) Ошибка «нет листа» возникает при попытке чтения информации с математического листа мб-, мд-файла, для которого не распределен физический лист.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситнетлиста*.

7) При отличном от нуля значении атрибута *максдлинстран* предпринимается ряд действий для разбиения ацпу-файла на логические страницы. В частности, введена ошибка «конец логической страницы», возникающая непосредственно после печати последней строчки текущей логической страницы.

Стандартная реакция: пустое действие, т. е. основное назначение данной ошибки состоит в том, чтобы дать пользователю возможность с помощью переопределения указать действия, которые необходимо выполнить при достижении конца логической страницы.

8) Ошибка «переполнение буфера» возникает в случае, если длина считываемого блока мл- или чпл-файла больше, чем длина массива (в частности,— буфера), в который производится считывание.

Стандартная реакция: пустое действие, т. е. основное назначение данной ошибки состоит в том, чтобы дать пользователю возможность с помощью переопределения указать действия, которые необходимо выполнить при возникновении указанных обстоятельств.

9) «Ошибка символа» возникает в случае, если значение атрибута *двоичный* для чпк-файла есть ложь и при чтении очередной перфокарты этого файла обнаружено, что какая-либо колонка содержит конфигурацию пробивок, не имеющую эквивалента в коде ДКОИ-8.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситошсмв*.

10) В случае сбоя в работе внешнего устройства система предпринимает ряд мер для перезапуска операции, во время выполнения которой произошел сбой. Если попытки перезапуска не заканчиваются удачей, то возникает «ошибка внешнего устройства».

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситошву*.

11) «Ошибка в параметрах обмена» возникает в том случае, когда нарушен интерфейс с системной процедурой, т. е. параметры операции обмена по типу, по числовой величине или по их количеству не соответствуют требованиям.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситошпарамобмена*.

12) Ошибка «аннулировать сообщение» возникает при нажатии на пишущей машинке клавиши «аннулирование». Нажатие этой клавиши завершает текущее сообщение. Отметим, что текст сообщения не ликвидируется автоматически и может быть считан обычным образом.

Стандартная реакция: пустое действие.

13) Ошибка «конец обмена» возникает в момент завершения обмена.

Стандартная реакция: пустое действие.

14) Ошибка «обмен ликвидирован» возникает при работе с аварийным объектом связи в случае, если операция обмена по тем или иным причинам отменяется (см. §§ 4, 5).

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситобменликвид*.

15) Ошибка «нет внешнего объекта» возникает при попытке доступа к ликвидированному внешнему объекту.

Стандартная реакция: выполняется структурный переход без параметров по динамической ситуации *ситнетвнеш*.

16) Ошибка «нет ресурса» возникает в случае, если в системе нет ресурсов, чтобы удовлетворить заявку на создание внешнего объекта, на внешнем устройстве указанного типа. Например, требуется создать мб файл, а на барабане нет свободного места.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситнетрес*.

17) Ошибка «привилегированный обмен» возникает в случае, если в непривилегированном режиме выполняются привилегированные операции с объектом.

Стандартная реакция: в случае, если в непривилегированном режиме производится обмен данными типа адресной информации, последние преобразуются к типу набор без изменения информационной части. В остальных случаях выполняется структурный переход по ситуации *ситпривобмен*.

18) Ошибка «нет в архиве» возникает в следующих случаях:

— когда в справочнике (или контексте) не найден элемент с данным именем;

— если при позиционировании справочника обнаружено начало или конец справочника.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситнетварх*.

19) Ошибка «нет архивной ссылки» возникает в случае, когда архивная операция, требующая непустой ссылки на объект, встречает пустую ссылку на внешний объект.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситнетархссл*.

20) «Ошибка в параметрах архивной операции» возникает в случае, когда нарушен интерфейс с архивными операциями.

Стандартная реакция: выполняется структурный переход по динамической ситуации *ситошпарамарх*.

21) «Ошибка атрибута» возникает при неверном задании атрибута открытия или атрибута внешнего объекта.

Стандартная реакция: выполняется структурный переход по ситуации *ситошатр*.

Таблица 2

Ошибка	1	2	3	4	5
Логический конец файла	р	с	*ситлкф*	*лкф*	*режлкф*
Логический конец контейнера	р	с	*ситлкк*	*лкк*	*режлкк*
Физический конец файла	р	с	*ситфкф*	*фкф*	*режфкф*
Физический конец контейнера	л	с	*ситфкк*	*фкк*	*режфкк*
Конец тома	р			*лкт*	*режлкт*
Нет листа	л	с	*ситнетлиста*	*нетлиста*	*режнетлиста*
Конец логической страницы	л	с	—	*клс*	*режклс*
Переполнение буфера	р	с	—	*переполнбуф*	*режпереполнбуф*
Ошибка символа	р	с	*ситошсмв*	*ошсмв*	*режошсмв*
Ошибка устройства	р	с	*ситошву*	*ошву*	*режошву*
Ошибка в параметрах обмена	л	с	*ситошпарамобмена*	*ошпарамобмена*	*режошпарамобмена*
Аннулировать сообщение	р	с	—	*аннул*	*режаннул*
Конец обмена	р	с	—	*конобмена*	*режконобмена*
Обмен ликвидирован	р	с	*ситобменликвид*	*обменликвид*	*режобменликвид*
Нет внешнего объекта		с	*ситнетвнеш*	*нетвнеш*	*режнетвнеш*
Нет ресурса		с	*ситнетрес*	*нетрес*	*режнетрес*
Привилегированный обмен	р	с	*ситпривобмен*	*привобмен*	*режприво бмен*
Нет в архиве			*ситнетварх*	*нетварх*	*режнетварх*
Нет архивной ссылки			*ситнетархслк*	*нетархслк*	*режнетархслк*

Окончание табл. 2

Ошибка	1	2	3	4	5
Ошибка в параметрах архивной операции			*ситошпарамарх*	*ошпарамарх*	*режошпарамарх*
Ошибка атрибута			*ситошатр*	*ошатр*	*режошатр*

В колонках табл. 2 приведена следующая информация:
1 — характер ошибки обмена (л — логическая ошибка, р — реальная),
2 — режим выполнения стандартной реакции (с — синхронный режим),
3 — идентификатор стандартной динамической ситуации,
4 — идентификатор атрибута, определяющего процедуру реакции на ошибку,
5 — идентификатор атрибута, определяющего режим выполнения реакции на ошибку.

* Ошибки обмена подразделяются на логические и реальные. Логические ошибки обнаруживаются системными процедурами в процессе формирования заявки на обмен. Реальные ошибки обнаруживаются внешними устройствами уже в процессе непосредственного выполнения операции. В табл. 2 для каждой ошибки указан ее характер и режим выполнения стандартной реакции.

2. Режим выполнения реакции на ошибку и стандартные ситуации

Инициирование выполнения реакции на ошибку может происходить в «логический момент времени», в «реальный момент времени» или «при синхронизации».

Под инициированием в логический момент времени понимается инициирование выполнения реакции во время выполнения процедуры синхронного обмена (буферизованного или непосредственного), или во время выполнения процедуры запуска параллельного обмена (буферизованного или непосредственного). В данном случае начало выполнения реакции синхронизовано с выполнением программы.

Под инициированием в реальный момент времени понимается случай, когда реальное выполнение операции на внешнем устройстве происходит параллельно с выполнением задачи, и выполнение реакции на ошибку инициируется во время этого реального выполнения. В данном случае начало выполнения реакции происходит асинхронно с выполнением задачи. Такой случай может возникнуть как в операциях параллельного обмена (буферизованного или непосредственного), так и в операциях синхронного буферизованного обмена. Последнее возможно как следствие того, что система осуществляет предварительное чтение блоков файла в буфер и запаздывающий сброс буферов на файл параллельно с выполнением программы.

Под инициированием при синхронизации понимается инициирование выполнения реакций в тот момент, когда задача осуществляет синхронизацию с процессом буферизованного или непосредственного параллельного обмена (например, путем выполнения операции ожидания открытия семафора завершения обмена). В данном случае начало выполнения реакции на ошибку синхронизовано с выполнением программы.

После того как выполнение реакции на ошибку инициировано, оно может происходить в синхронном, либо в параллельном режиме. В первом случае задача ожидает окончания выполнения реакции; во втором случае оба процесса выполняются параллельно. В табл. 3 приведена зависимость момента инициирования выполнения реакции от характера ошибки, режима выполнения операции обмена и режима выполнения реакции на ошибку.

Стандартные ситуации. Стандартная реакция аварийного характера состоит в следующем. Если в процессе выполнения какой-либо процедуры синхронного обмена (буферизованного или непосредственного) обнаружена логическая или реальная ошибка, или же если при выполнении процедуры запуска параллельного обмена обнаружена логическая ошибка, то выполнение реакции инициируется в логический момент времени. Это приводит к прекращению выполнения процедуры обмена в результате структурного перехода по стандартной динамической ситуации, соответствующей обнаруженной ошибке. Программа может определить реакцию на возникновение ситуации. Для этого нужно, чтобы место вызова процедуры синхронного обмена охватывалось (в общем случае динамически) структурным предложением, управляющим этой ситуацией. Например:

```
до * ситлкф
    (чит (ф1 , след , м1))
при
    ситлкф : запф ( печ , : ''конец файла'')
всесит
```

Если же такого динамически охватывающего структурного предложения нет, то выполнение независимого процесса прекращается и предпринимается конечная реакция на ситуацию, предусмотренная системой.

Если в процессе выполнения параллельного обмена обнаружена реальная ошибка, то выполнение стандартной реакции инициируется при синхронизации. Это значит, что структурный переход по соответствующей ситуации происходит не из места вызова процедуры запуска параллельного обмена, а из места, где выполняется операция синхронизации задачи с процессом обмена. Например, пусть в программе есть два ⟨структурных предложения⟩:

```
до * ситлкф
    (читпар (ввв1 , след , м1))
при
    ситлкф : ... R1 ...
всесит ;
```

```
...
до * ситлкф
   (ждать (читатр (ввв1 , смфобмена)))
при
   ситлкф : ... R2 ...
всесит
```

При возникновении ошибки «логический конец файла» будет выполняться действие R2, а не R1.

3. Переопределение стандартной реакции

Массив процедур реакций. Для переопределения стандартной реакции существует специальный объект, называемый массивом процедур реакций (МПР). Этот объект предназначен для хранения информации о переопределениях. МПР создается генератором объекта, который выдает указатель созданного МПР. Каждой ошибке обмена соответствуют два атрибута МПР: атрибут, определяющий процедуру обработки ошибки, и атрибут, определяющий режим выполнения этой процедуры. Мнемоника этих атрибутов приведена ранее в табл. 2.

Процедура реакции. Значением атрибута, определяющего процедуру реакции на некоторую ошибку, может быть:

1. Пустой объект формата ф64. В этом случае при возникновении данной ошибки выполняется стандартная реакция. Значение атрибута, определяющего режим выполнения реакции, игнорируется.

2. Процедура с одним параметром. В этом случае при возникновении данной ошибки в операции обмена выполняется вызов данной процедуры. В качестве фактического параметра ей передается аварийный объект связи (см. § 4). Значение второго атрибута определяет режим выполнения процедуры обработки ошибки.

Таким образом, для переопределения стандартной реакции на некоторую ошибку необходимо изменить соответствующий атрибут МПР так, чтобы его значением стала метка нужной процедуры реакции. Например:

```
ошпозп1 := мпр(лкф : проц(авоб)
(запф(печ, : "конец файла"); ложь));
```

Здесь переменной *ошпозп1* присваивается МПР с переопределенной реакцией на ошибку «логический конец файла».

Режим выполнения процедуры реакции. Атрибут, определяющий режим выполнения реакции на ошибку, может принимать следующие значения: 0 — синхронный режим; 1 — параллельный режим (процедура реакции будет запускаться как параллельный процесс на вновь созданном стеке); 2 — параллельный режим, возможный только в привилегированных процедурах (в этом случае процедура реакции будет запускаться как «попутная работа» на каком-либо из стеков, существующих в данный момент в системе).

Если атрибут, определяющий процедуру реакции, установлен, а режим выполнения не определен, то по умолчанию выбирается синхронный режим.

Связывание МПР. МПР можно использовать двумя способами:

1. Указатель МПР может являться значением атрибута *ошобмена* оперативного объекта связи с внешним объектом. В этом случае все указанные в МПР переопределения имеют силу при любой операции обмена, осуществляемой с помощью данного объекта связи. Например, после выполнения модификации атрибута

запатр (*позп*1. *ошобмена* : *ошпозп*1)

для всех операций, осуществляемых с помощью позиционной переменной *позп*1, будут иметь силу переопределения, указанные в МПР *ошпозп*1.

2. Указатель МПР может быть подан в качестве последнего параметра в операцию обмена. В этом случае указанные в нем переопределения имеют силу только при выполнении данной операции. Например:

чит(*позп*1, *след*, *ошобмена* : *ошпозп*1)

4. Операции над аварийным объектом связи

Фактическим параметром процедуры реакции на ошибку является аварийный объект связи с файлом. Этот объект предназначен для того, чтобы идентифицировать (а) объект, с помощью которого осуществлялся доступ к файлу, и (б) конкретную операцию, при выполнении которой произошла ошибка («аварийная операция»).

В случае мб-, мд-файла операции, следующие за аварийной, выполняются обычным образом. В случае файлов других типов временно приостанавливается реальное выполнение последующих операций обмена с данным файлом. Логическое выполнение операций не приостанавливается, в результате чего к данному файлу может накопиться очередь заявок на чтение/запись.

Над аварийным объектом связи определены следующие операции:

1) *испавобмен* (*авоб*)

Эта процедура производит попытку синхронного перезапуска аварийной операции. Если попытка прошла успешно, то в качестве результата выдается **истина**, и выполнение всех приостановленных операций (если таковые есть) возобновляется уже в параллельном режиме. Если при перезапуске опять возникла ошибка, то рекурсивного вызова реакции на ошибку не происходит, выдается **ложь**, и выполнение приостановленных операций, если таковые есть, не возобновляется.

2) *ликвидавобмен* (*авоб*)

Эта процедура фиктивно завершает аварийную операцию, т. е. реальная операция с внешним устройством отменяется, но логически операция завершается с ошибкой «обмен ликвидирован». Выполнение приостановленных операций, если таковые есть, не возобновляется. Для их возобновления можно воспользоваться операцией *испавобмен*.

3) *ликвидавочрд(авоб)*

Эта процедура фиктивно завершает аварийную операцию и все приостановленные операции, если таковые есть.

Атрибуты аварийного объекта. Аварийный объект связи имеет ряд собственных атрибутов, которые можно запрашивать. В основном эти атрибуты идентифицируют аварийные оперативные объекты связи. С аварийными объектами можно делать те же операции, что и с обычными. Отличие заключается в том, что если такой объект подан в операцию обмена, то операция не попадает в очередь приостановленных операций, но выполняется обычным образом. Атрибуты аварийного объекта перечислены ниже.

авфайл — значением этого атрибута является аварийный заголовок открытого файла.

авбвв — значением этого атрибута является аварийный блок ввода/вывода. Этот БВВ соответствует аварийной операции, т. е. запрашивая его атрибуты, можно получить всю информацию об операции, при выполнении которой возникла ошибка.

авпозп — значением этого атрибута является аварийная позиционная переменная.

автбуф — значением этого атрибута является аварийная таблица буферов.

5. Завершение выполнения реакции

Процедура реакции на ошибку должна выдавать логическое значение **истина** или **ложь**. Если результат — **истина** и аварийная операция и(или) приостановленные операции не выполнены и не ликвидированы, то система выполняет операцию *испавобмен*. Если результат — **ложь** и аварийная операция и(или) приостановленные операции не выполнены и не ликвидированы, то система выполняет операцию *ликвидавочрд*.

Таблица 3

Режим реакции	Реальная ошибка		Логическая ошибка
	Синхронный режим операции	Параллельный режим операции	
Синхронный	л* / л	с* / с	л / л
Параллельный	р** / л	р*** / р	л / л

В табл. 3 изображена зависимость момента инициирования выполнения реакции от характера ошибки, режима операции и режима реакции.

В клетках в числителе обозначен момент инициирования для случая буферизованного обмена, а в знаменателе — для случая непосредственного обмена. В табл. 3 использованы следующие обозначения: л — логический момент времени; р — реальный момент времени; с — при синхронизации; * — реакция возможна только на ошибку, возникшую при чтении или при позиционировании; ** — ошибка может возникнуть при предварительном чтении блока файла или при запаздывающем сбросе буфера; *** — ошибка может возникнуть как при реальном параллельном выполнении операции, так и при предварительном чтении блока файла или при запаздывающем сбросе буфера (последние два действия неявно выполняются системой, когда программа пользуется последовательным буферизованным обменом). *

Приложение 1

КОДЫ ЛИТЕР

В таблице указаны числовые величины литер. Приведены величины в десятичном и шестнадцатеричном представлении.

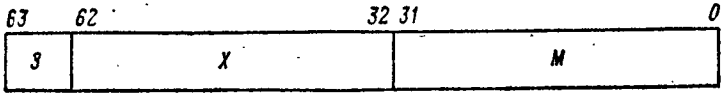
Символ	Число		Символ	Число		Символ	Число	
	10	16		10	16		10	16
Пробел	64	40	Ю	184	B8	Я	221	DD
[	74	4A	Б	186	BA	S	226	E2
. (точка)	75	4B	Ц	187	BB	T	227	E3
<	76	4C	Д	188	BC	U	228	E4
(	77	4D	Ф	190	BE	V	229	E5
+	78	4E	Г	191	BF	W	230	E6
!	79	4F	А	193	C1	X	231	E7
&	80	50	В	194	C2	Y	232	E8
]	90	5A	С	195	C3	Z	233	E9
¤	91	5B	D	196	C4	У	235	EB
*	92	5C	Е	197	C5	Ж	236	EC
)	93	5D	F	198	C6	Ь	238	EE
;	94	5E	G	199	C7	Ы	239	EF
^	95	5F	Н	200	C8	0	240	F0
— (минус)	96	60	I	201	C9	1	241	F1
/	97	61	И	203	CB	2	242	F2
_	106	6A	Й	204	CC	3	243	F3
, (запятая)	107	6B	Л	206	CE	4	244	F4
%	108	6C	J	209	D1	5	245	F5
_ (подчерк)	109	6D	К	210	D2	6	246	F6
>	110	6E	L	211	D3	7	247	F7
?	111	6F	М	212	D4	8	248	F8
:	122	7A	N	213	D5	9	249	F9
#	123	7B	О	214	D6	Э	250	FA
@	124	7C	Р	215	D7	Ш	251	FB
' (апостроф)	125	7D	Q	216	D8	Э	252	FC
=	126	7E	R	217	D9	Щ	253	FD
" (кавычка)	127	7F	П	220	DC	Ч	254	FE

Приложение 2

ВНУТРЕННЕЕ РАСПАКОВАННОЕ ПРЕДСТАВЛЕНИЕ (ВРП) ОБЪЕКТОВ

Для каждого типа приведено изображение соответствующего ⟨тега⟩ (если оно есть в языке) и числовая величина тега. С помощью X обозначаются неиспользуемые разряды.

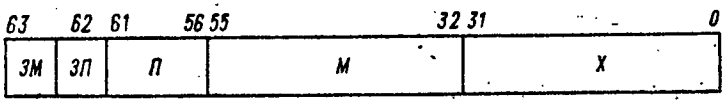
1. Тип: целое 32, цел32, 62.



З — знак числа, М — мантисса,

Примечание. После считывания из памяти разряды X заполнены нулями; при записи не используются.

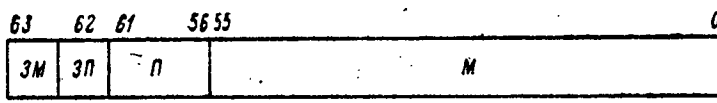
2. Тип: вещественное 32, вещ32, 57.



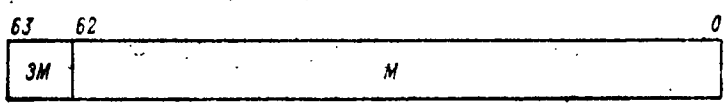
ЗМ — знак мантиссы, ЗП — знак порядка,
П — порядок (шестнадцатеричный).

Примечание. После считывания из памяти разряды X заполнены нулями; при записи не используются.

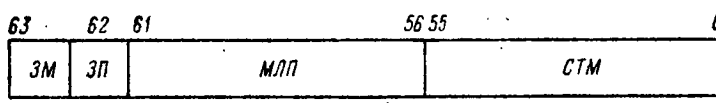
3. Тип: вещественное 64, вещ64, 33.

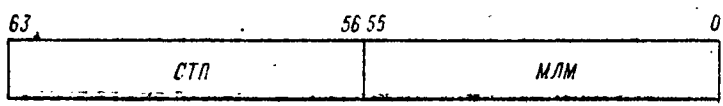


4. Тип: целое 64, цел64, 34.



5. Тип вещественное 128, вещ128, 37.

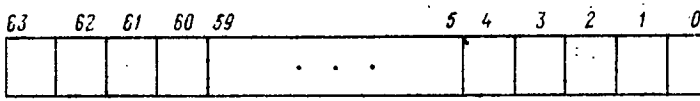




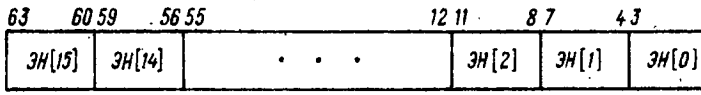
СТМ — старшие разряды мантиссы,
МЛМ — младшие разряды мантиссы,
СТП — старшие разряды порядка,
МЛП — младшие разряды порядка.

6. Тип: набор, наб, 20.

Битовый набор

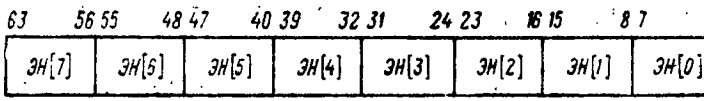


Цифровой набор



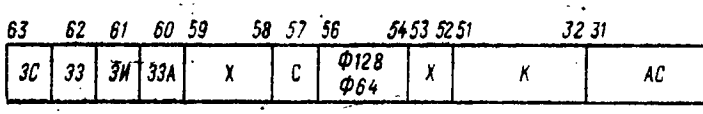
ЭН[i] — i-й элемент набора.

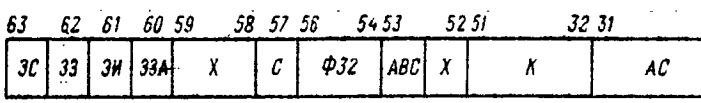
Литерный набор

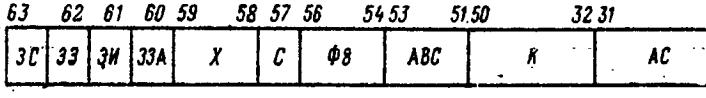


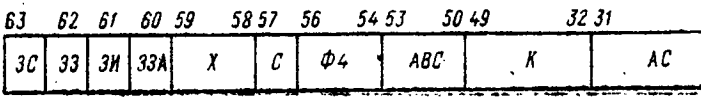
7. Тип: указатель с-вектора, деск, 0.

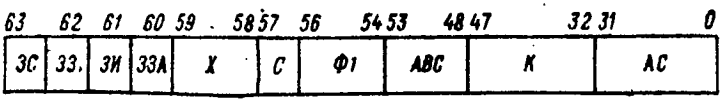
Приведены пять случаев для различных значений формата элементов вектора.





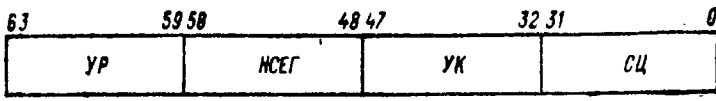






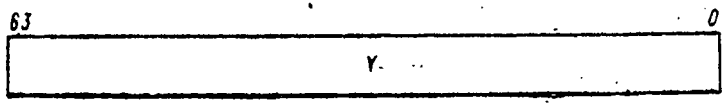
ЗС — запрет считывания из массива,
ЗС — запрет записи в массив,
ЗИ — запрет исполнения массива, как программного кода,
ЗЗА — запрет записи в массив адресной информации,
С — дескриптор описывает массив в стеке,
К — количество элементов,
АС — адрес слова,
АВС — адрес внутри слова.

8. Тип: метка процедуры, мпроц, 10.



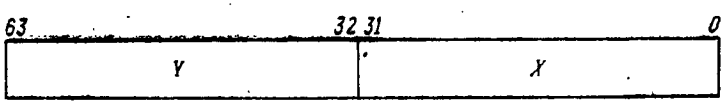
УР — лексикографический уровень процедуры,
НСЕГ — номер программного сегмента процедуры,
УК — указатель команды входа в процедуру.
СЦ — контекст идентификаторов для данной процедуры.

9. Тип: пусто 64, п64, 32.



Y — произвольное заполнение.

10. Тип: пусто 32, п32, 56.

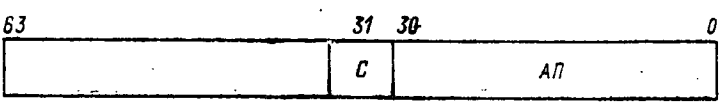


Примечание. После считывания из памяти разряды X заполнены нулями; при записи не используются.

11. Тип: указатель объекта связи, 7.

Поля указателя распределяются операционной системой.

12. Тип: семафор, 5.

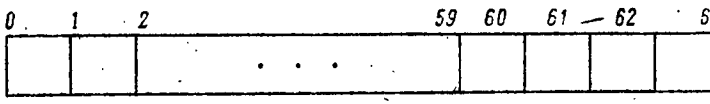


С — семафорный бит (0 — открыт, 1 — закрыт),
АП — адрес начала очереди ждущих процессов.

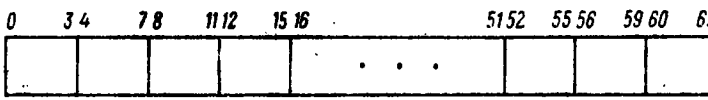
Приложение 3

ПОЗИЦИИ ЭЛЕМЕНТОВ ВНУТРИ СЛОВА ПАМЯТИ

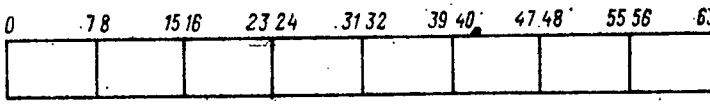
1. Элементы формата ф1.



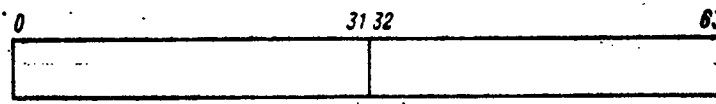
2. Элементы формата ф4.



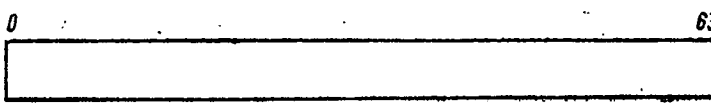
3. Элементы формата ф8.



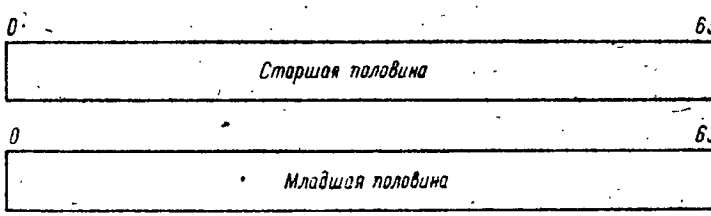
4. Элементы формата ф32.



5. Элемент формата ф64.



6. Элемент формата ф128.



Элемент размещается в двух подряд идущих словах памяти. Адрес младшей половины на 1 больше адреса старшей половины.

Приложение 4

СТАНДАРТНЫЕ КАРТЫ

Метки пк-файла. Начальная и конечная метка пк-файла — это перфокарты, у которых пробивки в колонках 61 — 80 имеют стандартную конфигурацию (см. макеты 1 и 2).

В колонках 1 — 3 карты начальной (конечной) метки пробито слово НАЧ (КОН). В колонках 4 — 60 карты начальной метки пробита в текстовом виде информация об атрибутах пк-файла. Колонки распределены следующим образом:

5 — 9: слово, задающее значение атрибута *двоичный* (КПК12 — **ложь**, ДВОИЧ — **истина**).

11 — 27: строка, задающая атрибут *имяфайла*. Если в этих колонках пробелы, то файл считается непомеченным.

29—34: слово, задающее значение атрибута *типфайла* (ДАННЫЕ — 0, ТЕКСТ — 3).

36—43: строка, задающая атрибут *имяяз*. Если в этих колонках пробелы, то значение атрибута полагается равным пустому набору, т. е. содержимое файла рассматривается как произвольный текст.

Карта пользователя помещается в пакетном задании следом за начальной меткой пк-файла. На этой карте пробит текст, содержащий имя пользователя, и, возможно, его пароль. Текст имеет вид

П = А ... А ККП

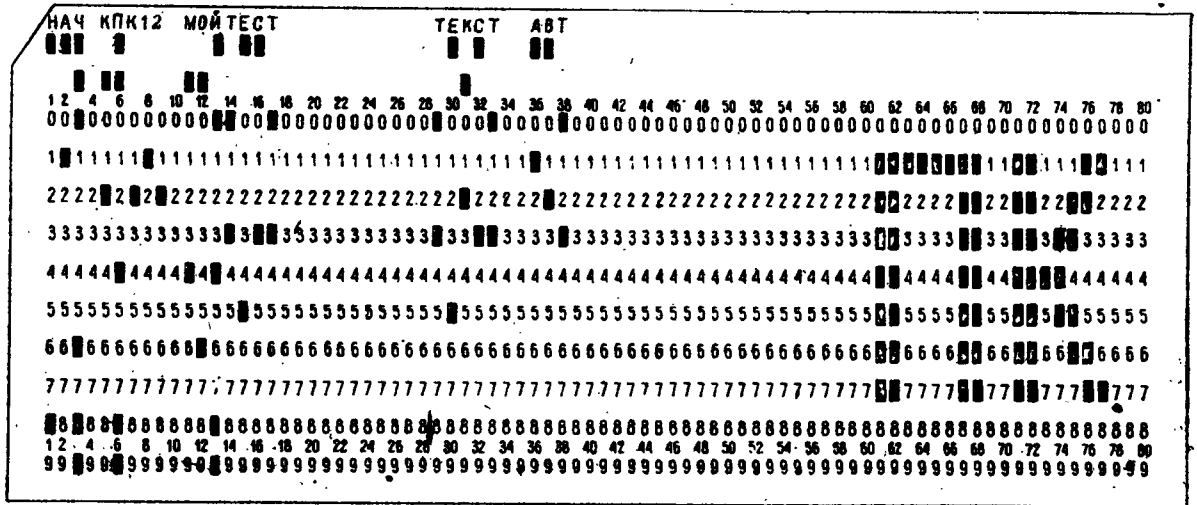
или

П = А ... А П = В ... В ККП

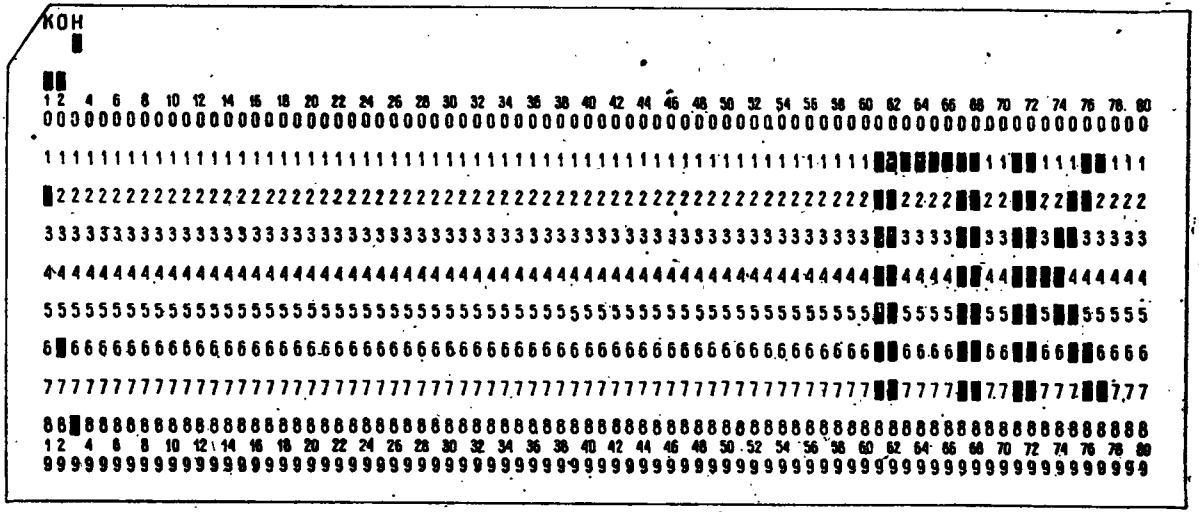
где А ... А и В ... В — соответственно литеры имени пользователя и литеры пароля (см. макет 3). Реально карта пользователя может занимать несколько перфокарт. На последней из них пробито слово ККП (Конец Карты Пользователя).

Карта пользователя может содержать только слово ККП. В этом случае программа задания не имеет собственного внешнего контекста и может использовать только объекты глобального внешнего контекста, доступного всем пользователям.

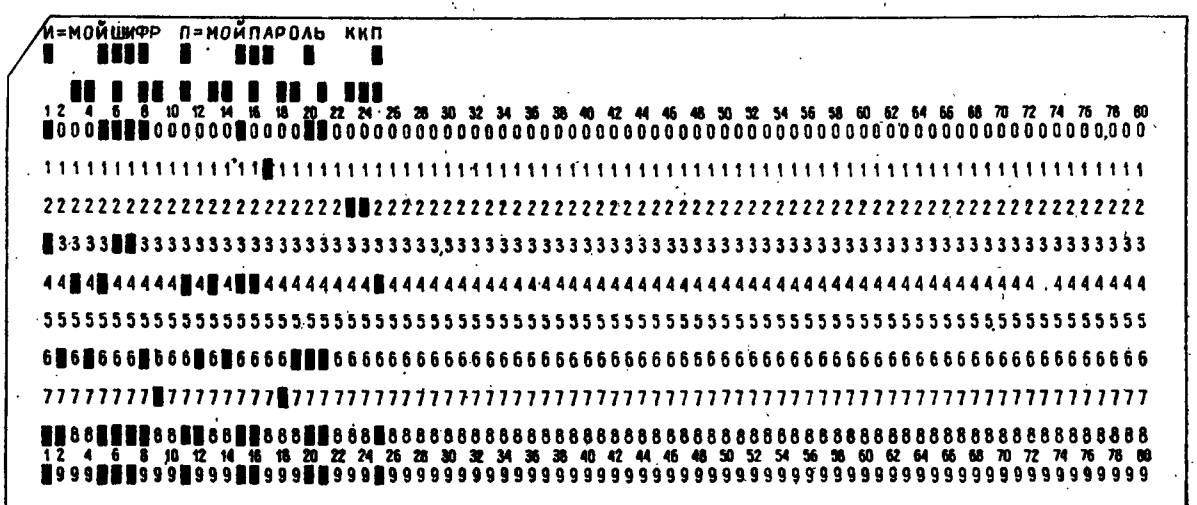
Система обеспечивает режим, когда с некоторых устройств могут вводиться пакеты заданий, в которых каждое индивидуальное задание не содержит карты пользователя. Задания таких пакетов могут использовать только объекты глобального контекста.



Макет 1. Начальная метка пк-файла.



Макет 2. Конечная метка пк-файла.



Макет 3. Карта пользователя.

Приложение 5

СИНТАКСИС ЯЗЫКА

В этом приложении собраны синтаксические правила языка. Каждой группе правил предшествует номер и название параграфа или пункта, в котором вводится эта группа.

Глава 2.

2.2. Лексические элементы.

⟨буква⟩ ::= A | B | C | D | E | F | G | H | I | J | K
| L | M | N | O | P | R | S | T | U | V | W | X | Y
| Y | Z | Ю | Б | Ц | Д | Ф | Г | Й | Л | Я | Ч | Ж | Ы
| Ж | Ы | Ь | З | Ш | Щ | Э | Ъ | У

⟨цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
⟨двоичная цифра⟩ ::= 0 | 1
⟨восьмеричная цифра⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
⟨шестнадцатеричная цифра⟩ ::=
0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D |
E | F

⟨литера⟩ ::= ⟨буква⟩ | ⟨цифра⟩ | ⟨вертикальная черта⟩
| ⟨кавычка⟩ | ⟨пробел⟩ | [|] | < | > | + | −
| ; | : | . | / | ! | % | , | (|) | = | - | '
| @ | ¤ | # | & | ^ | *

⟨идентификатор⟩ ::= ⟨буква⟩{⟨буква или цифра⟩}...
⟨буква или цифра⟩ ::= ⟨буква⟩ | ⟨цифра⟩
⟨целое⟩ ::= ⟨цифра⟩...
⟨вещественное⟩ ::=
⟨целое⟩.⟨целое⟩{⟨порядок⟩} | ⟨целое⟩⟨порядок⟩
⟨порядок⟩ ::= e{⟨знак⟩}⟨целое⟩
⟨знак⟩ ::= + | −
⟨поэлементное представление⟩ ::=
1"⟨двоичная цифра⟩..." | 3"⟨восьмеричная цифра⟩..."
| 4"⟨шестнадцатеричная цифра⟩..." | 8"⟨литера⟩..."
⟨формат⟩ ::= ⟨простой формат⟩ | ⟨строчный формат⟩
⟨простой формат⟩ ::= **ф32** | **ф64** | **ф128**
⟨строчный формат⟩ ::= **ф1** | **ф4** | **ф8**

3. Изображения.

⟨изображение⟩ ::=
⟨число⟩ | ⟨набор⟩

| ⟨константный вектор⟩ | ⟨текст процедуры⟩
| ⟨константный файл⟩ | ⟨константный справочник⟩
| ⟨пустой объект⟩

3.1. Изображение числа.

⟨число⟩ ::=
{ ⟨тег целого⟩}⟨целое⟩
| { ⟨тег вещественного⟩}⟨вещественное⟩
| ⟨тег вещественного⟩⟨целое⟩
| ⟨тег числа⟩⟨поэлементное представление⟩
| ⟨тег⟩
⟨тег целого⟩ ::= **цел32** | **цел64**
⟨тег вещественного⟩ ::= **вещ32** | **вещ64** | **вещ128**
⟨тег числа⟩ ::= ⟨тег целого⟩ | ⟨тег вещественного⟩
⟨тег⟩ :: = ⟨тег числа⟩ | **наб** | **мпроц** | **деск** | **миход** | **п32** | **п64**

3.2. Изображение набора.

⟨набор⟩:: = ⟨поэлементное представление⟩ ... | **истина** | **ложь** | ""

3.3. Пустой объект.

⟨пустой объект⟩ ::= **пусто32** | **пусто64**

4. Описание.

⟨описание⟩ :: =
⟨описание простых переменных⟩ | ⟨описание базы⟩
| ⟨описание простых констант⟩ | ⟨описание меток⟩
| ⟨описание процедур-констант⟩ | ⟨описание полей⟩
| ⟨описание статических ситуаций⟩ | ⟨описание текстов⟩

5. Простые константы.

⟨описание простых констант⟩ ::=
конст ⟨список идентификации констант⟩
⟨идентификация константы⟩ : : =
⟨идентификатор⟩ = ⟨выражение⟩

6.1. Простые переменные.

⟨описание простых переменных⟩ ::=
⟨простой формат⟩⟨список идентификации переменных⟩
⟨идентификация переменной⟩ ::=
⟨идентификатор⟩{:=⟨выражение⟩}
| ⟨идентификатор⟩=⟨идентификатор⟩

6.2. Массив.

⟨генератор массива⟩ ::=
⟨генератор в-массива⟩ | ⟨генератор с-вектора⟩
⟨генератор в-массива⟩ ::=
⟨локализация⟩[⟨список выражений⟩]⟨формат⟩
⟨генератор с-вектора⟩ ::=
⟨локализация⟩**вект**[⟨выражение⟩{:⟨выражение⟩}]
⟨формат⟩
⟨локализация⟩ ::= **лок** | **глоб**
⟨константный вектор⟩::= ⟨массив констант⟩ | ⟨строка⟩
⟨массив констант⟩ ::=
мконст⟨формат⟩(⟨список элементов массива констант⟩)
⟨элемент массива констант⟩ ::=
⟨число повторений⟩⟨выражение⟩
⟨число повторений⟩(⟨список элементов массива констант⟩)
⟨число повторений⟩ ::= /⟨целое⟩/
⟨строка⟩ ::=
стр1 ⟨поэлементное представление⟩ ...
| **стр4** ⟨поэлементное представление⟩ ...
| **стр8** ⟨поэлементное представление⟩...
⟨элемент массива⟩ ::=
⟨первичное⟩[{⟨формат⟩}⟨список выражений⟩]
⟨подмассив⟩::= ⟨первичное⟩ [{⟨формат⟩} ⟨диапазон⟩]
⟨диапазон⟩ ::= {⟨выражение⟩} : {⟨выражение⟩}

6.3. Описание и использование полей.

⟨поле переменной⟩ ::= ⟨переменная⟩.⟨описатель поля⟩
⟨описатель поля⟩ ::=
⟨непосредственный описатель поля⟩
| ⟨идентификатор⟩
⟨непосредственный описатель поля⟩ ::=
[{⟨строчный формат⟩}⟨выражение⟩{: {⟨выражение⟩}}]
⟨поле значения⟩ ::= ⟨первичное⟩.⟨описатель поля⟩
⟨формирование⟩ ::=
⟨первичное⟩.(⟨список элементов формирования⟩)
⟨элемент формирования⟩ ::=
⟨описатель поля⟩ : ⟨выражение⟩ | **тип**:⟨выражение⟩
⟨описание поля⟩ ::= **поле**⟨список идентификаций полей⟩
⟨идентификация поля⟩ ::=
⟨идентификатор⟩ = ⟨описатель поля⟩

6.4. Указатель переменной.

⟨формирование указателя⟩ ::=@⟨переменная⟩
⟨указуемая переменная⟩ ::= ⟨первичное⟩@

6.5. Переменная.

⟨переменная⟩ ::=
⟨идентификатор⟩ | ⟨элемент массива⟩
| ⟨поле переменной⟩ | ⟨указуемая переменная⟩
| ⟨закрытая переменная⟩ | ⟨подстановка текста⟩

7. Выражения.

⟨выражение⟩ ::=
⟨формула⟩ | ⟨присваивание⟩ | ⟨операции над массивами⟩

7.1. Формула.

⟨формула⟩ ::=
⟨логическая сумма⟩{**экв**⟨логическая сумма⟩}...
⟨логическая сумма⟩ ::=
⟨логическое произведение⟩
{**или**⟨логическое произведение⟩}...
⟨логическое произведение⟩ ::= ⟨отношение⟩
{**и**⟨отношение⟩}...
⟨отношение⟩::=
⟨сумма⟩{⟨операция отношения⟩⟨сумма⟩}...
| ⟨сравнение строк⟩
⟨сумма⟩ ::=
⟨произведение⟩
{⟨операция типа сложения⟩⟨произведение⟩}...
⟨произведение⟩ ::=
⟨сомножитель⟩{⟨операция типа умножения⟩
⟨сомножитель⟩}...
⟨сомножитель⟩ ::=
⟨одноместная формула⟩{⟨операция типа степени⟩
⟨одноместная формула⟩}...
⟨одноместная формула⟩ ::=
{⟨одноместная операция⟩}⟨первичное⟩
| ⟨формирование указателя⟩
⟨первичное⟩ ::=
⟨изображение⟩ | ⟨идентификатор⟩
| ⟨элемент массива⟩ | ⟨подмассив⟩
| ⟨указуемая переменная⟩ | ⟨вызов⟩
| ⟨генератор⟩ | ⟨закрытое выражение⟩
| ⟨формирование паспорта⟩ | ⟨формирование⟩
| ⟨форматный обмен⟩ | ⟨признак⟩
| ⟨подстановка текста⟩ | ⟨запрос атрибута⟩
| ⟨модификация атрибута⟩ | ⟨внешнее имя⟩
| ⟨двоичный обмен⟩ | ⟨поле значения⟩

7.2. Приоритет операций.

⟨операция отношения⟩ ::= ⟨операция сравнения⟩ | <=> | **среди**
⟨операция сравнения⟩ ::= <> | = | > | >= | < | <=
⟨операция типа сложения⟩ ::= + | − | +: | −: | **плюс** | **минус**
⟨операция типа умножения⟩ ::= * | **умнд** | *: | /: | **остат** -
⟨операция типа степени⟩ ::= **
⟨одноместная операция⟩ ::=

целокр	**целобр**	**вещокр**	**вещобр**
ф32окр	**ф32обр**	**ф64окр**	**ф64обр**
цел64окр	**цел64обр**	**в128**	**целзн**
стнаб	**млнаб**	**естьнабор**	**естьпусто**
естьцел	**естьвещ**	**естьф32**	**естьф64**
естьф128	**не**	**пче**	**перв1**
знак	**длина**	**адрес**	**тип**
дес	**бит**		

7.11. Генератор.

⟨генератор⟩ ::=
⟨генератор массива⟩ | ⟨генератор ситуации⟩
| ⟨генератор объекта связи⟩ | ⟨генератор паспорта⟩
| ⟨генератор внешнего объекта⟩

8. Присваивание.

⟨присваивание⟩ ::=
⟨переменная⟩ := ⟨выражение⟩
| ⟨переменная⟩ := : ⟨выражение⟩

9. Закрытое предложение.

⟨закрытое предложение⟩ ::=
{блок} ⟨небазированное закрытое предложение⟩
⟨небазированный закрытый оператор⟩ ::=
⟨замкнутый оператор⟩ | ⟨условный оператор⟩
| ⟨выбирающий оператор⟩ | ⟨цикл⟩
| ⟨структурный оператор⟩
⟨небазированное закрытое выражение⟩ ::=
⟨замкнутое выражение⟩ | ⟨условное выражение⟩
| ⟨выбирающее выражение⟩ | ⟨структурное выражение⟩
⟨небазированная закрытая переменная⟩ ::=
⟨замкнутая переменная⟩ | ⟨условная переменная⟩
| ⟨выбирающее переменную⟩
⟨последовательное предложение⟩ ::=

{⟨описание⟩;} ... {⟨помеченный оператор⟩;} ...
⟨помеченное предложение⟩

⟨помеченное предложение⟩ ::= {⟨метка⟩} ... ⟨предложение⟩

⟨оператор⟩ ::=

⟨присваивание⟩	⟨модификация атрибутов⟩
⟨вызов⟩	⟨генератор внешнего объекта⟩
⟨структурный переход⟩	⟨запуск задачи⟩
⟨операция над массивами⟩	⟨подстановка текста⟩
⟨закрытый оператор⟩	⟨двоичный обмен⟩
⟨форматный обмен⟩	⟨формирование паспорта⟩
⟨переход⟩	

9.1. Замкнутое предложение.

⟨замкнутое предложение⟩ ::=
(⟨последовательное предложение⟩)
| **начало**⟨последовательное предложение⟩ **конец**

9.2. Условное предложение.

⟨условное предложение⟩ ::=
если ⟨условие⟩ **то** ⟨последовательное предложение⟩
{**инес** ⟨условие⟩ **то** ⟨последовательное предложение⟩} ...
{**иначе** ⟨последовательное предложение⟩}
все

⟨условие⟩ ::=
{⟨описание⟩;} ... {⟨помеченный оператор⟩;} ...
⟨помеченное выражение⟩

9.3. Выбирающее предложение.

⟨выбирающее предложение⟩ :: =
выбор ⟨вычисление номера⟩ **из**
⟨список размеченных альтернативных предложений⟩
{**иначе**⟨последовательное предложение⟩}
всевыб
| **выбор** ⟨вычисление номера⟩ **из**
⟨список последовательных предложений⟩
иначе ⟨последовательное предложение⟩
всевыб

⟨вычисление номера⟩ ::=
{⟨описание⟩;} ... {⟨помеченный оператор⟩;} ...
⟨помеченное выражение⟩

⟨размеченное альтернативное предложение⟩ ::=
{⟨номер⟩:,} ... ⟨номер⟩ : ⟨последовательное предложение⟩

⟨номер⟩ ::= ⟨выражение⟩

9.4. Цикл.

⟨цикл⟩ ::=
{⟨заголовок цикла⟩}
цикл
{⟨описание⟩;} ...
{⟨помеченный оператор⟩;} ...
⟨помеченный оператор⟩
повторить

⟨заголовок цикла⟩ ::=
для⟨идентификатор⟩{**от**⟨выражение⟩}
{⟨компонента-до⟩}
| **от**⟨выражение⟩{⟨компонента-до⟩}

⟨компонента-до⟩ ::= **до**⟨выражение⟩ | **вниздо** ⟨выражение⟩

10.1. Структурное предложение.

⟨структурное предложение⟩ ::=
до⟨список определений ситуаций⟩
⟨закрытое предложение⟩
{**при**⟨список альтернативных предложений⟩**всесит**}

⟨определение ситуации⟩ ::= ⟨идентификатор⟩ | * ⟨идентификатор⟩

⟨описание статических ситуаций⟩ ::=
статсит⟨список идентификаторов⟩

⟨генератор ситуации⟩ ::=
ситуация ({⟨список установок атрибутов⟩})

⟨альтернативное предложение⟩ ::=
{⟨идентификатор⟩:,} ... ⟨идентификатор⟩
{({⟨описание формальных⟩;} ... ⟨описание формальных⟩)}
:⟨последовательное предложение⟩

⟨описание формальных⟩ ::= {⟨простой формат⟩}
⟨список идентификаторов⟩

10.2. Структурный переход.

⟨структурный переход⟩ ::= ⟨первичное⟩!
{(⟨список выражений⟩)}

11.1. Процедура.

⟨текст процедуры⟩ ::=
проц({⟨параметры⟩})⟨закрытый оператор⟩
| **функция** ({⟨параметры⟩})⟨закрытое выражение⟩

⟨параметры⟩ ::=
{⟨описание формальных процедуры⟩;} ...
⟨описание формальных процедуры⟩

⟨описание формальных процедуры⟩ ::=
⟨описание формальных⟩ | ⟨описание базы⟩
⟨описание процедур-констант⟩ ::=
процедура ⟨список идентификации процедур⟩
⟨идентификация процедуры⟩ ::=
⟨идентификатор⟩{=⟨текст процедуры⟩}
⟨вызов⟩ ::= ⟨первичное⟩({⟨список фактических⟩})
⟨фактический⟩ ::=
{пак32}⟨выражение⟩
| **прог**⟨выражение⟩
| **имя прог** ⟨переменная⟩
| **имя**⟨переменная⟩

11.3. Задача.

⟨запуск задачи⟩ ::=
задача (⟨список установок атрибутов⟩)
⟨текст программы⟩{(⟨список фактических⟩)}

12. Программы.

⟨текст программы⟩ ::=
программа {⟨описание⟩;} ... {⟨оператор⟩;} ...
⟨выражение⟩**конец**
| ⟨изображение⟩
| ⟨закрытый оператор⟩

13.1. Операции форматного обмена.

⟨форматный обмен⟩ ::=
запф(⟨выражение⟩, ⟨список элементов вывода⟩)
| *читф*(⟨выражение⟩, ⟨список элементов ввода⟩)
| *запфм*(⟨массив обмена⟩, ⟨список элементов вывода⟩)
| *читфм*(⟨массив обмена⟩, ⟨список элементов ввода⟩)
⟨элемент вывода⟩ ::=
⟨выражение⟩{:⟨формат обмена⟩} | :⟨позиционирование⟩
⟨элемент ввода⟩ ::=
имя⟨переменная⟩{:⟨формат обмена⟩}
| ⟨выражение⟩{:⟨формат обмена⟩}
| :⟨позиционирование⟩
⟨массив обмена⟩ ::= ⟨выражение⟩ | **имя**⟨переменная⟩

13.3. Позиционирование.

⟨позиционирование⟩ ::= ⟨литерал⟩{⟨размещение⟩} ...
| ⟨размещение⟩ ...
⟨размещение⟩ ::= {⟨повторитель⟩}⟨код размещения⟩
{⟨литерал⟩}
⟨литерал⟩ ::=
{⟨повторитель⟩"⟨литера⟩..."} ...

{⟨повторитель⟩}"⟨литера⟩"
⟨код размещения⟩ ::= *x* | *y* | *κ* | *l* | *p*
⟨повторитель⟩ ::=⟨целое⟩ | *n*(⟨выражение⟩)
здесь *n* — латинское *N*.

13.4. Обмен, управляемый форматом.

⟨формат обмена⟩::= {⟨вставка⟩}⟨трафарет⟩
⟨трафарет⟩ ::=
| ⟨трафарет числа⟩ | ⟨трафарет целого⟩
| ⟨трафарет вещественного⟩ | ⟨трафарет литерного⟩
| ⟨трафарет логического⟩ | ⟨трафарет двоичного⟩
| ⟨трафарет тегированного⟩
⟨вставка⟩ ::= ⟨литера⟩{⟨смещение⟩}... | ⟨смещение⟩...
⟨смещение⟩ ::= {⟨повторитель⟩}*x*{⟨литерал⟩}
⟨трафарет числа⟩ ::=
{⟨повторитель⟩}*g*{(⟨выражение⟩ {,⟨выражение⟩
{,⟨выражение⟩}})}{⟨вставка⟩}
⟨трафарет целого⟩ ::= {⟨поле знака⟩}⟨поле цифр⟩
⟨поле знака⟩ ::=
⟨знак⟩{⟨вставка⟩}⟨поле плавающего знака⟩
{⟨вставка⟩}
⟨поле цифр⟩ ::= ⟨подполе цифр⟩...
⟨подполе цифр⟩ ::=
{⟨повторитель⟩}{*s*}*d*{⟨вставка⟩}
| {⟨повторитель⟩}*z*{⟨условная вставка⟩}
| {⟨повторитель⟩}*z*⟨вставка⟩
⟨поле плавающего знака⟩ ::=
⟨подполе плавающего знака⟩...⟨знак⟩
⟨подполе плавающего знака⟩ ::=
{⟨повторитель⟩}*f*{⟨условная вставка⟩}
| {⟨повторитель⟩}*f*⟨вставка⟩
⟨условная вставка⟩ ::= *c*⟨вставка⟩
⟨трафарет вещественного⟩ ::=
⟨трафарет с фиксированной точкой⟩
| ⟨трафарет с плавающей точкой⟩
⟨трафарет с фиксированной точкой⟩ ::=
⟨трафарет целого⟩⟨поле точки⟩⟨поле цифр⟩
⟨трафарет с плавающей точкой⟩ ::=
⟨трафарет с фиксированной точкой⟩⟨поле порядка⟩
| ⟨трафарет целого⟩⟨поле порядка⟩
⟨поле точки⟩ ::= {*s*}.{⟨вставка⟩}
⟨поле порядка⟩ ::= {*s*}*e*⟨трафарет целого⟩
⟨трафарет логического⟩ ::= *b*{⟨вставка⟩}
⟨трафарет литерного⟩ ::=⟨подполе литер⟩...
⟨подполе литер⟩ ::= {⟨повторитель⟩}{*s*}*a*{⟨вставка⟩}

⟨трафарет двоичного⟩ ::= ⟨основание⟩r⟨поле цифр⟩
{⟨вставка⟩}
⟨основание⟩ ::= 1 | 3 | 4
⟨трафарет тегированного⟩ ::= ⟨основание⟩t{⟨вставка⟩}

14. Операции над массивами.

⟨операции над массивами⟩ ::=
⟨пересылка⟩ | ⟨поиск⟩ | ⟨упаковка⟩

14.1. Пересылка.

⟨пересылка⟩ ::=
⟨вектор назначения⟩⟨операция пересылки⟩
⟨источник пересылки⟩
{&&⟨источник пересылки⟩}
⟨источник пересылки⟩ ::=
⟨безусловная пересылка⟩ | ⟨условная пересылка⟩
| ⟨заполнение⟩
| ⟨перевод⟩ | ⟨распаковка⟩
⟨вектор назначения⟩ ::= ⟨формула⟩ | мод⟨переменная⟩
⟨операция пересылки⟩ ::= <:=
⟨безусловная пересылка⟩ ::=
{⟨простой формат⟩}⟨вектор источника⟩**длиной**
⟨максколичество⟩
⟨условная пересылка⟩ ::=
⟨безусловная пересылка⟩**пока**
⟨индикатор отношения⟩⟨первичное⟩
⟨вектор источника⟩ ::= ⟨первичное⟩ | мод⟨переменная⟩
⟨максколичество⟩ ::= ⟨первичное⟩ | мод⟨переменная⟩
⟨индикатор отношения⟩ ::= ⟨операция сравнения⟩
| **среди** |**не среди**
⟨заполнение⟩ ::=
{⟨простой формат⟩}**заполн**⟨первичное⟩
{**длиной**⟨максколичество⟩}
⟨перевод⟩ ::=
перевод⟨вектор источника⟩{**длиной**⟨максколичество⟩}
по⟨первичное⟩
⟨распаковка⟩ ::=
⟨операция распаковки⟩⟨первичное⟩
{**длиной**⟨максколичество⟩}
⟨операция распаковки⟩ ::= **распак** | **распакзн**

14.2. Операция поиска.

⟨поиск⟩ ::= ⟨поиск по строке⟩ | ⟨поиск по маске⟩
⟨поиск по строке⟩ ::=
⟨указатель источника⟩ **от**⟨условие поиска⟩

⟨условие поиска⟩ ::=
⟨индикатор отношения⟩⟨первичное⟩
| ⟨максколичество⟩{**либо**⟨индикатор отношения⟩ .
⟨первичное⟩}
⟨указатель источника⟩ ::= ⟨формула⟩ | **мод**⟨переменная⟩
⟨поиск по маске⟩ ::=
поиск(⟨выражение⟩,⟨выражение⟩{,{⟨выражение⟩,}
⟨выражение⟩})
⟨способ выборки⟩⟨операция сравнения⟩⟨первичное⟩
⟨способ выборки⟩ ::= **до** | **вниздо** | **спискомдо** | **цепьюдо**

14.3. Сравнение строк.

⟨сравнение строк⟩ ::=
⟨вектор источника⟩⟨операция сравнения⟩
/⟨вектор источник⟩
{**длиной**⟨максколичество⟩}
⟨операция сравнения⟩ ::= < | > | = | <> | <= | >=

14.4. Упаковка.

⟨упаковка⟩ ::= **пак**⟨вектор источник⟩
{**длиной**⟨максколичество⟩}

14.5. Признаки.

⟨признак⟩ ::= **тго** | **тгп** | **тги**

15.1. Формирование паспорта.

⟨формирование паспорта⟩ ::=
формавм({⟨описатель⟩}[⟨список идентификаторов⟩]=
⟨описатель⟩[⟨список диапазонов или индексов⟩])
⟨описатель⟩ ::= ⟨идентификатор⟩ | ⟨закрытое выражение⟩
⟨диапазон или индекс⟩ ::= ⟨суперпозиция⟩{=⟨диапазон⟩}
⟨суперпозиция⟩ :: =⟨слагаемое⟩{⟨знак⟩⟨слагаемое⟩}...
⟨слагаемое⟩ ::= ⟨сомножитель⟩{*⟨сомножитель⟩}

15.3. Генератор паспорта.

⟨генератор паспорта⟩ ::=
⟨локализация⟩[{,}...]
| ⟨локализация⟩[⟨список выражений⟩]

16. Участок базированной области.

⟨описание базы⟩ ::= **база**⟨идентификатор⟩

17. Текстовые макросы.

⟨описание текстов⟩ ::= **текст**⟨список идентификации текстов⟩
⟨идентификация текста⟩ ::=

⟨идентификатор⟩{({⟨список идентификаторов⟩})}
=⟨предложение⟩
⟨подстановка текста⟩ ::=
⟨идентификатор⟩{({⟨список предложений⟩})}
| ⟨идентификатор⟩[⟨список предложений⟩]

18. Атрибуты объектов.

⟨запрос атрибута⟩ ::=
читатр (⟨выражение⟩, {⟨уточнение⟩,} ⟨выражение⟩)
⟨уточнение⟩ ::= ⟨выражение⟩
⟨модификация атрибутов⟩ : : =
запатр(⟨выражение⟩,{⟨уточнение⟩,}
⟨список установок атрибутов⟩)
⟨установка атрибута⟩ ::= ⟨выражение⟩ : ⟨выражение⟩

19. Описание и использование меток.

⟨описание меток⟩::= **метка**⟨список идентификаторов⟩
⟨метка⟩::= ⟨идентификатор⟩ ^:
⟨переход⟩::= на⟨одноместная формула⟩

Глава 3.
2.1. Генератор объекта.

⟨генератор внешнего объекта⟩::=
⟨спецификация внешнего⟩({⟨выражение⟩,}⟨выражение⟩
{,⟨список установок атрибутов⟩})
⟨спецификация внешнего⟩: : =
генфайл | *венконтейнер* | *венспр*
⟨генератор объекта связи⟩: : =
⟨спецификация объекта связи⟩
({⟨выражение⟩,} ⟨список установок атрибутов⟩)
| ⟨спецификация объекта связи⟩
({⟨список установок атрибутов⟩})
⟨спецификация объекта связи⟩: : =
файл | *позп* | *тбуф* | *бвв* | *контейнер* | *прогр* | *мпр*
| *генув*

5.4. Внешнее имя.

⟨внешнее имя⟩ ::= {⟨идентификатор⟩}⟨план поиска⟩
⟨план поиска⟩ : : = ⟨многослоговое имя⟩ | // [⟨выражение⟩]
⟨многослоговое имя⟩ ::= ⟨слог⟩...
⟨слог⟩ ::= //⟨буква или цифра⟩... | //⟨стандартный слог⟩
⟨стандартный слог⟩ ::= *.обкт* | *.текст* | *.код* | *.псевдо*

7. Обмен. Общие сведения

⟨двоичный обмен⟩ ::=
⟨идентификатор операции⟩(⟨выражение⟩,
⟨спецификация адреса⟩{,⟨выражение⟩}
{,*ошобмена*:⟨выражение⟩})
⟨идентификатор операции⟩ :: =
чит | *читпар* | *зап* | *заппар* | *нов* | *уст* | *устпар*
⟨спецификация адреса⟩ ::=
⟨выражение⟩
| ⟨направление⟩{(⟨выражение⟩)}
| ⟨выражение⟩, ⟨направление⟩{(⟨выражение⟩)}
| *нач* | *кон* |*канал*(⟨выражение⟩)
⟨направление⟩ ::= *след* | *пред*

13. Изображение файла.

⟨константный файл⟩ ::=
⟨изображение произвольного файла⟩
| ⟨изображение текстового файла⟩
⟨изображение произвольного файла⟩ : : =
файл⟨вложенный файл⟩
⟨изображение текстового файла⟩ : : =
тфайл{(⟨список установок атрибутов⟩)}
⟨определение ограничителя⟩⟨текст файла⟩
⟨ограничитель⟩

14. Константный справочник

⟨изображение справочника⟩ ::=
спрконст(⟨список элементов справочника⟩)
⟨элемент справочника⟩ ::=⟨идентификатор⟩{@}=
⟨выражение⟩

ЛИТЕРАТУРА

1. Бабаян Б. А., Сахин Ю. Х. Система Эльбрус.— Программирование, 1980, 6.
2. Пентковский В. М. Основные характеристики автокода МВК Эльбрус. М.: ИТМ и ВТ, 1977.
3. Пентковский В. М. Средства структурированного программирования для языка высокого уровня.— М.: ИТМ и ВТ, 1976.
4. Пратт Т. Языки программирования: разработка и реализация.— М.: Мир, 1979, с. 60—61.
5. Лондон Р., Шоу М., Вульф А. У. Абстракция и верификация в альфарде: пример таблицы идентификаторов./ В кн.: Создание качественного программного обеспечения.— Новосибирск: ВЦ СО АН СССР, 1978.
6. Дюар Р. Б. К., Гранд А., Лю С.-К., Шонберг Э., Шварц Дж. Т. Сетл как инструмент построения качественного программного обеспечения./В кн.: Создание качественного программного обеспечения.— Новосибирск: ВЦ СО АН СССР, 1978.
7. Вейнгаарден А. и др. Пересмотренное сообщение об алголе-68.— М.: Мир, 1979.
8. Дал О. И., Мюрхаут Б., Нюгард К., Симула67. Универсальный язык программирования.— М.: Мир, 1968.
9. Лавров С. С., Силигадзе Г. С. Автоматическая обработка данных. Язык лисп и его реализация.— М.: Наука, 1979.
10. Клещев А. С., Темов В. Л. Язык программирования инф и его реализация.— Л.: Наука, 1973.
11. Strachey S. Varieties of programming language, International Computer State of the Art Report, report 7, Higher level languages, 1972.
12. Wirth N. Pascal user manual and report.— Berlin: Springer Verlag, 1975.
13. Burstall R. M. et al. Programming in POP-2.— Edinburg university press, 1971.
14. Reynolds Y. C. GEDANKEN — a simple typeless language based on principle of completeness and reference concept.— CACM, 1970, 13, 5.
15. Thurber K. J., Myrna J. W. System design of a cellular APL computer.— IEEE transactions on computers, C-19, 4, 1970.

16. Wirth N., Weber H. EULER: A generalisation of algol and its formal definition.— CACM, 1966, 9, 1.
17. Liscov B. et al. Abstraction mechanisms in Clu.— CACM, 20, 8.
18. Wirth N. Modula: a language for modular multiprogramming.— Software, Practice and Eperience, 1977, 7, 1.
19. Lampson B. W., Horning J. J. et al. Report on the programming language Euclid.— SIGPLAN Notices, 1977, 12, 2.
20. Shaw M., Wulf W. et al. Tartan-language design for the Ironmen requirements.— SIGPLAN Notices, 1978, 13, 9.
21. Ichbiah J. D. et al. Preliminary ADA reference manual (part A) and Rationale for the design of the ADA programming language (part B). SIGPLAN Notices, 1979, 14, 6.
22. Wulf An introduction to the construction and verification in Alphard.— IEEE transaction and software engineering, 1976, SE-2,4,
23. Conradi R., Holager P. A study of Mary's data types in a systems programming application/B кн.: Proc. of the IFIP Work. conf. on machine oriented higher level languages.— North Holland, 1974.
24. Cowan R. M. Burroughs B6700/B7700 work flow language/ B кн.: Proc. of the IFIP Work. conf. on command languages.— North Holland, 1975.
25. Dolotta T. A., Mashey J. B. Using a command language as the primary programming tool (UNIX shell)/B кн.: Proc. of the IFIP Work. conf. on command languages.— North Holland, 1980.
26. Madsen J. Highlights of CCL/B кн.: Proc. of the IFIP Work. conf. on command languages.— North Holland, 1980.
27. Lyle Don M. A hierarhy of higher order languages for systems programming/B кн.: Proc. of SIGPLAN symp. on lanquages for systems implementation.— SIGPLAN Notices 1971, 6, 9.
28. Van der Poel (ed.) The description of the Buddy Algorithm in some MOL's. Machine oriented higher level languages.— North Holland, 1974.
29. Ichbiah J. P. et al. The two-level approach to data defenition and space management in the Lis system. Proc. of ACM SIGPLAN — SIGOPS Interface meeting.— SIGPLAN Notices, 1973, 8, 9.
30. Sapper G. R. The programming language PS440 as a tool for implementing a timesharing system.— SIGPLAN Notices, 1971, 6, 9.
31. Wulf W. et al. BLISS — a banguage for systems programming.— CACM, 1971, 14, 12.
32. Berry D. M. Introduction to Oregano, ACM SIGPLAN Proc. of a symp. on data structures in programming languages.— Univ. of Florida, 1971.
33. Liskov B., Snyder A. Exception handling in Clu.— IEEE trasaction on software engineering. 1979, SE-5, 6.
34. Глушков В. М., Михновский С. Д., Рабинович З. Л. ЭВМ со структурной реализацией языков высокого уровня.— Кибернетика, 1981, 4.

СПИСОК ИДЕНТИФИКАТОРОВ СТАНДАРТНОГО КОНТЕКСТА

СОКРАЩЕНИЯ

А — атрибут
ВПНС — встроенная процедура нестандартного синтаксиса
ВПСС — встроенная процедура стандартного синтаксиса
ИНК — идентификатор нестандартной константы
ИСК — идентификатор стандартной константы
ЛК — литерная константа
СС — стандартная ситуация
ССВИ — стандартный слог внешнего имени

СПИСОК СИНТАКСИЧЕСКИХ КОНСТРУКЦИЙ

СПИСОК СЛУЖЕБНЫХ СЛОВ

СПИСОК ТЕХНИЧЕСКИХ ТЕРМИНОВ

Владимир Мстиславович Пентковский
АВТОКОД ЭЛЬБРУС. ПРИНЦИПЫ ПОСТРОЕНИЯ ЯЗЫКА И РУКОВОДСТВО К ПОЛЬЗОВАНИЮ

Редакторы: *Н. Н. Васина, Г. А. Слепнева*
Техн. редактор: *С. Я. Шкляр*
Корректоры: *О. А. Бутусова, Т. С. Вайсберг*

ИБ № 11865
Сдано в набор 19.01.82. Подписано к печати 08.09.82. Т-16758. Формат 84×108$^{1}/_{32}$. Бумага тип. № 3. Обыкновенная гарнитура. Высокая печать. Условн. печ. л. 18,48. Уч.-изд. л. 21,56. Тираж. 16 000 экз. Заказ № 28. Цена 1 р. 30 к.

Издательство «Наука»
Главная редакция
физико-математической литературы
117071, Москва, В-71, Ленинский проспект, 15

4-я типография издательства «Наука»
630077, Новосибирск, 77, Станиславского, 25