Б. МЕЙЕР, К. БОДУЭН

МЕТОДЫ ПРОГРАММИРОВАНИЯ 1

Перевод с французского
Ю. А. ПЕРВИНА

под редакцией
А. П. ЕРШОВА

Издательство «Мир» Москва 1982

ББК 32.973
М 45
УДК 681.142.2

М45 Мейер Б., Бодуэн К.

Методы программирования: В 2–х томах. Т.1. Пер. с франц. Ю.А. Первина. Под ред. и с предисловием А. П. Ершова.–М.: Мир, 1982 356 с.

Монография французских ученых, в которой систематически излагаются основные понятия информатики, обсуждаются трудные проблемы методологии программирования, дается сравнение известных языков программирования: ФОРТРАНа, АЛГОЛа W, ПЛ/1 и др. Изложение сопровождается упражнениями (с решениями).

В русском переводе книга разбита на два тома. В первый том (гл. I–V) наряду с общими сведениями о программировании рассмотрены основные свойства базовых языков программирования и управляющие структуры, а также обсуждаются подпрограммы и структуры данных.

Книга рассчитана на профессиональных программистов, желающих овладеть современными методами программирования. Может служить учебным пособием по языкам программирования и алгоритмам.

Редакция литературы по математическим наукам

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕИЗДАНИЯ

С момента выхода книги прошло уже 30 лет, но, перечитывая ее, понимаешь, что классика бессмертна. Всё, что здесь написано, актуально и по сей день, и будет актуально, пока существует программирование.

В далёком 1983-м году я, 18-тилетним парнем, пришел работать в одно ПКБ. Это была эра большим машин (ЕС ЭВМ, если еще кто помнит). Реальный мир программирования оказался слишком далёким от того, что преподавали в институтах. Оказалось, что на производстве не перемножали и даже не складывали матрицы! Зато мне выдали распечатки одного САПР на ФОРТРАНе, которые весили килограмм 5–6. В них то я и углубился. Программы были написаны в стиле того времени – идеально выровнены по 7-й колонке, без единого комментария и со всякими интересными трюками, как то GOTO во внутрь цикла и т.п. САПР был не очень популярный, спроса на него не было, и что бы заполнить свободное время, я решил заняться самообразованием и заглянул в техническую библиотеку ПКБ. И среди всевозможных справочников, задачников и учебников по ФОРТРАНу я случайно наткнулся на «Методы программирования».

Эта книга перевернула моё представление о программировании. Всё встало на свои места. Программирование вместо какого–то шаманства превратилось в строгую математическую дисциплину, где действуют свои законы. Находясь под сильным впечатлением от книги и не имея доступа к алголоподобным языкам (на тот момент использовались ФОРТРАН IV, КОБОЛ и ПЛ/1), я даже написал препроцессор для ФОРТРАНа на самом же ФОРТРАНе, который реализовывал все базовые управляющие структуры, описанные здесь (скажу сразу, что символьная обработка на ФОРТРАНе IV – занятие не для слабонервных).

В начале 90-х дни ПКБ были сочтены, пора было менять работу. Книга осталась в библиотеке. После этого я неоднократно возобновлял безуспешные попытки найти ее в продаже. Со второй половины 90-х я стал разыскивать ее в Интернете. В лучшем случае можно было оформить предварительный заказ (без гарантии, что он будет исполнен) на каком-нибудь книжном сайте. И на запрос: «Мейер Б. Бодуэн К. Методы программирования» поисковые службы выдавали тысячи ссылок, но все они были либо сообщениями в форумах, либо (что наиболее распространено) ссылками на литературу в университетских курсах.

Книга на русском языки вышла в 1982 году тиражом всего в 30 000 экземпляров (что для СССР очень мало). Да еще очень ненадёжный переплёт (кто держал эту книгу в руках знают). Так что к этому времени, очевидно, осталось очень мало живых экземпляров.

Судя по всему, переизданием этой книги с 82-го года не занималось ни одно издательство. Сейчас книжные магазины ломятся от литературы по информатике, но, к сожалению, там очень проблематично найти книги, подобные этой. Зато масса книг типа «Освой Java за 21 день», «C/C++ для хакеров», «Delphi изнутри» и т.д., где тщательно рассматриваются возможности получить доступ к private полям классов, создания «круглых» форм и прочие трюки, которые, по большому счету, к программированию не имеют никакого отношения. А вот книги подобной этой, которые закладывают самые основы программирования, встретить на прилавках книжных магазинов весьма проблематично.

Мне повезло, я все таки достал эту книгу. Так как издательства совершенно не интересуют фундаментальные труды, я решил восполнить этот пробел, издать ее в электронном виде и сделать доступной всему программистскому сообществу.

Работа над книгой была нелёгкой – всякий, кому хотя бы раз приходилось сканировать, распознавать, форматировать и кое–где исправлять 700 страниц текста с таблицами, примерами и массой математических формул, поймёт меня. Сначала я хотел отделаться минимумом – оставить все в таком формате, в каком был оригинал. Но проведя несколько дней в безуспешных попытках подобрать размеры шрифтов, отступов и межстрочных интервалов, я принял решение переформатировать книгу в формат A4. Минимальные изменения коснулись расположения некоторых рисунков. Так же я позволил себе по ходу дела исправить кое–какие ошибки и неточности, которые, очевидно, были внесены уже в процессе перевода и вёрстки. К середине второго тома устал не только я, но, судя по всему, и те, кто выпускал книгу в 82-м году. Так, если до этого неточности заключались, например, в пропущенной точке с запятой или двоеточия, то дальше стали выпадать небольшие куски кода, для восстановления которых приходилось тщательно штудировать материал. Надеюсь, читатель простит меня, если обнаружится какая-то незамеченная мной опечатка или неточность после распознавания – физически очень сложно выверить каждое предложение, каждое слово.

декабрь 2007

QUIDAM
quidam2007@mail.ru

ПРЕДИСЛОВИЕ РЕДАКТОРА ПЕРЕВОДА

Разрыв между последними достижениями информатики и повседневной практикой программирования, между трудностью задачи программирования и примитивностью доступного арсенала средств, между головокружительным прогрессом оборудования и инертностью программного обеспечения, по–видимому, нарастает с каждым годом. Именно это положение вещей обсуждалось во время дискуссии, посвященной проблеме подготовки квалифицированных программистов 80-х годов, которая стала одним из наиболее ярких событий недавно прошедшего в Японии и Австралии Всемирного конгресса ИФИП–80. Признавая кризисное состояние практики программирования и демонстрируя широкий спектр мнений по поводу актуальной проблематики, участники дискуссии были, однако, единодушны в своей уверенности в «светлом будущем программирования»; они пришли к заключению, что достичь его можно лишь путем тщательно подготовленного всеобщего и интенсивного курса обучения, не только охватывающего сумму технических приемов, но в определенном смысле формирующего саму личность программиста.

Естественно, что этот тезис родился не вчера. Уже в течение нескольких лет преподаватели программирования настойчиво ищут лучшие формы построения такого курса, охватывающего современные методологические положения информатики. В этой серии хотя еще и не окончательных, но в целом идущих в. нужном направлении попыток достойное место занимает предлагаемая вниманию читателей монография Б. Мейера и К. Бодуэна. Она опирается на уже вошедшие в практику преподавания концепции многоязыкового или, правильнее сказать, надъязыкового подхода к обучению программированию. Используя в качестве своего рода языка спецификаций неформально описанные языковые конструкции «высокого уровня», авторы весьма подробно показывают, как эти конструкции реализуются в распространенных алгоритмических языках, выбрав из них типичные представители: ФОРТРАН, ПЛ/1 и АЛГОЛ W. Авторы сумели, не выходя из контекста реального языка, внушить читателю оптимизм в его стремлении научиться хорошо составлять программы, не дожидаясь создания «идеальных» языков программирования.

Центральными главами учебника являются гл. III–VI, в которых авторы излагают основные управляющие структуры, структуры данных и правила процедурной декомпозиции алгоритмов, уделяя при этом особое внимание рекурсивному программированию как основному средству установления связи между структурой программы и структурой данных и упаковки неограниченной последовательности действий в конечный текст программы. Обращает на себя внимание употребление авторами циклов в качестве вторичной конструкции, возникающей как реализация инвариантного соотношения или как инструмент повышения эффективности программы при переходе от рекурсивного описания к итеративному. Весьма полезным для читателя будет подробное изучение способов связи фактических данных вызовов процедур с формальными параметрами описаний процедур. Авторы пытаются, и, в целом, не без успеха, внедрить в сознание программиста понятия абстрактных типов данных, инвариантов и других логических утверждений о программе как естественно возникающие понятия, неотъемлемые от процесса грамотного программирования.

Все это авторы делают, не скрывая своего личного отношения к положительным и отрицательным сторонам повседневной практики программиста, с которой они, судя по всему, знакомы не понаслышке. Изложение материала живое и подчас темпераментное; оно требует от читателя чувства юмора и способности к критическому анализу. Интересна активная позиция авторов, направленная на преодоление американской

традиции, доминирующей сейчас во многих аспектах программирования – от терминологии до методологических положений.

Книга несвободна от некоторых литературных и композиционных недостатков, очевидно обусловленных большим объемом, нехваткой времени да и отсутствием сложившихся форм изложения материала. Было бы однако излишним педантизмом фиксировать на них внимание читателя. В целом же нет сомнений, что книга Б. Мейера и К. Бодуэна найдет внимательных читателей, которым она принесет немало пользы.

Академгородок,
ноябрь 1980 г. *А. П. Ершов*

ПРЕДИСЛОВИЕ

В этой книге представлены некоторые современные методы программирования для электронных вычислительных машин. Она ориентирована главным образом на читателей двух категорий – на студентов, желающих углубить свои знания в таких областях, как языки программирования, алгоритмы, структуры данных, и на профессиональных программистов, которые хотят лучше освоить свой предмет.

В течение последних нескольких лет информатика – исследование автоматической обработки информации, и в частности проблем, связанных с принципами и применениями ЭВМ, – мало–помалу формируется в самостоятельную науку, которая объединяет несколько более или менее независимых дисциплин. Одна из таких дисциплин – программирование (область, в недавнем прошлом еще мало исследованная и подвластная лишь «фанатикам битов и микросекунд») – стала предметом многочисленных систематических исследований и достигла определенной степени формализации.

Наряду с попытками исследователей построить теорию программирования возникло новое явление, несколько умерившее энтузиазм, с которым пользователи ЭВМ встречали бурное развитие возможностей вычислительных машин. Значительное увеличение скорости вычислений и емкости запоминающих устройств многим давало надежду, что они смогут решать задачи постоянно растущих размеров и сложности; иллюзия подкреплялась еще тем, что программирование поначалу не было очень сложным делом: инженер или служащий мог быстро, не получая специальной подготовки по информатике, выучить необходимые ему в практической деятельности основные элементы такого языка, как ФОРТРАН или КОБОЛ, и «прокрутить» несколько простых программ. По мере того как распространялась иллюзия простоты программирования, положение дел существенно усложнялось; пользователи, реализуя тот или иной проект, приобретали достаточно горький опыт, сталкиваясь с «порогом сложности», за которым ощущалась невозможность контролировать создаваемые объекты или процессы.

На большие проекты в области информатики влияют не только размеры и мощность ЭВМ и соответствующего оборудования – процессоров, периферийных устройств, линий связи и т.д., того, что называют техническим обеспечением. Определенные ограничения накладывает и «человеческий фактор»: написание программ, их отладка, использование, непрерывная адаптация к изменениям окружающей обстановки – все, что называют математическим обеспечением. Ограничения математического обеспечения, как и человеческого мышления, фундаментальны и ни в коей мере не снимаются прогрессом технического обеспечения.

Итак, становится ясным, что именно эта вторая компонента определяющим образом обусловливает успех или неудачу проекта и определяет самую существенную часть его стоимости. Весьма ощутимы временные и экономические потери, порождаемые ошибками замысла, некачественной документацией, использованием неадекватных языков программирования или незнанием основ техники алгоритмов. Красноречива американская статистика по этому поводу: если размеру большой программы поставить в соответствие время, необходимое для ее реализации, и число занятых в ней программистов, то окажется, что все происходит так, словно каждый программист создает 5 команд в день, а остальное время пишет команды, которые потом признаются бесполезными или неправильными, и разыскивает сделанные ошибки.

Существуют, однако, методы, позволяющие нарушить печальный цикл –

«ошибка–исправление–новая ошибка», хорошо знакомый тем, кто пытался когда-нибудь запустить программу на ЭВМ. Научные исследования привели не к созданию какого–то чудодейственного средства, а к некоторым принципам программирования, благодаря которым производительность труда программистов можно значительно повысить. «Производительность» здесь не означает «эффективность» в традиционном смысле слова (т.е. искусство и приемы, позволяющие сэкономить байт или полсекунды); речь идет о понятии более общем, которое подразумевает в первую очередь «правильность» создаваемых программ, т.е. что они решают действительно ту задачу, для которой были задуманы (нынешние «операционные» программы в этом смысле преподносят немало сюрпризов); что они легко отлаживаются и тестируются, легко читаются и могут передаваться другому программисту; что они используют лучшие возможности машин, которые их исполняют. «Производительность», таким образом, означает здесь оптимальное использование и машин, и людей.

Упомянутые выше научные разработки зачастую находили лишь весьма слабое отражение в практической деятельности. Прежде всего эти разработки действительно казались несколько абстрактными: часто они опирались на языки программирования, мало распространенные в «реальном мире» и выглядевшие гипотетическими; рассматриваемые примеры иной раз были оторваны от повседневной практики вычислительных центров; наконец, их нормативный и догматический тон отталкивал профессионалов–практиков.

Главный постулат, принятый в нашей книге, состоит в том, что одни и те же задачи ставятся как в научном, так и в производственном аспекте. Наша цель – убедить читателя той и другой ориентации в том, что программа на ФОРТРАНе может быть «хорошей» или «плохой». Даже будучи убежденными в достоинствах, например, СИМУЛЫ 67, мы полагаем, что абсурдно изгонять из программистского рая подавляющее большинство пользователей, не имеющих доступа к транслятору с этого языка. Наша книга представляет собой компромисс: мы пытались быть точными, не позволяя себе углубляться в чисто теоретические рассуждения.

В нашем изложении мы будем опираться в основном на три языка: АЛГОЛ W, ФОРТРАН и ПЛ/1. Первый из них дает хороший образец ясности в описании алгоритмов, второй является наиболее распространенным языком для научных и инженерных задач, третий стремится объединить возможности числовой обработки и действий с данными, которыми оперируют в задачах управления производством. Мы будем также эпизодически обращаться к некоторому анонимному языку уровня ассемблера; наконец, принципы и алгоритмы будут введены с помощью более абстрактной алгоритмической нотации, приведенной в приложении А.

Нас могут упрекнуть в таком выборе; попытаемся оправдать его, остановившись вкратце на исключенных нами из рассмотрения языках. Наиболее важный из них – КОБОЛ, язык, может быть, еще более, чем ФОРТРАН, распространенный в «реальном мире» информатики. Он обладает оригинальными особенностями с точки зрения структурирования данных. Однако жесткость его управляющих структур и многословие, к которому он вынуждает программиста, делают изнурительным описание всякого сколько-нибудь сложного алгоритма. С другой стороны, самые интересные из его качеств унаследованы языком ПЛ/1. Среди других оставленных нами в стороне языков – ЛИСП, который, несмотря на свой теоретический универсализм, имеет ограниченное применение (основные идеи ЛИСПа, однако, вводятся в гл. V и VI); более глубокое, чем у нас, изучение ассемблеров привело бы к акцентированию внимания на какой–нибудь конкретной машине и усложнило введение алгоритмических понятий. Наконец, в семействе АЛГОЛа, которое дает лучшие образцы «структурированных» языков, мы могли бы выбрать СИМУЛУ 67, АЛГОЛ 68 или ПАСКАЛЬ, у каждого из которых свои особые достоинства. Мы остановились на АЛГОЛе W, используемом во многих университетах. Он сохранил основные черты

своего «предка» – АЛГОЛа 60 и вместе с тем вводит важнейшие понятия, представленные в языках последнего времени, в частности структурирование данных. При этом язык остается простым и легко изучаемым.

Разумеется, эта книга не является учебником по ФОРТРАНу, АЛГОЛу W или ПЛ/1, и библиография адресует читателя к специальным пособиям. Тем не менее в ней рассмотрены основные свойства этих языков, достаточные для обычного программирования: базовые – в гл. II, другие – по мере того, как они становятся необходимыми или позволяют проиллюстрировать важную задачу. Мы хотели таким образом избежать путаницы, которая могла бы возникнуть в результате конкурирующего употребления трех языков; мы стремились также дать возможность читателю, знающему например, ФОРТРАН, но не знакомому ни с АЛГОЛом W, ни с ПЛ/1, на простых примерах приобщиться к тем преимуществам, которые открывают эти два последних языка. Научиться разбирать новый язык программирования относительно просто и всегда полезно для тех, кто уже знает другой язык.

Следует заметить, что описание свойств языков программирования занимает относительно много места, несомненно больше, чем мы этого хотели, потому что некоторые свойства объясняются непросто. Существует некоторая тенденция, пытающаяся полностью освободить курс программирования от изучения языков: это отрицательная реакция на появление многочисленных учебников, посвященных введению в программирование, которые зачастую представляют собой только описание того или иного языка и основное внимание в которых уделяется нетипичным свойствам этого языка; не желая возвращаться к крайностям прежних времен, мы считаем, однако, опасным не дать программисту возможности овладеть основным орудием – его средством выражения. Важно, например, описать критерии, позволяющие выделять «разумные» подмножества языка–то, что мы делаем в этой книге по отношению к ФОРТРАНу, АЛГОЛу W и ПЛ/1, – и вынести сравнительные суждения о различных языках. Можно заметить также, что сами спорные свойства языка часто отражают наличие лежащей в их основе принципиальной проблемы.

После первой главы, знакомящей с начальными понятиями информатики и программирования, во второй главе вводятся основные элементы обычных языков программирования, в частности ФОРТРАНа, АЛГОЛа W и ПЛ/1, а затем даются основы алгоритмической нотации высокого уровня, которая используется во всей оставшейся части книги для разъяснения конструкций и описаний программ. В гл. III рассматриваются управляющие структуры, т.е. основные схемы, которые позволяют описать ход автоматических вычислений. В гл. IV обсуждается одно из центральных понятий в информатике – подпрограммы; акцент здесь сделан на проблемах информационной связи подпрограмм и на методах управления локальными данными, здесь же вводится понятие сопрограммы. Гл. V посвящена изложению понятия структур данных и описанию важнейших из них (стеки, файлы, деревья, графы и т.п.); делается попытка с особой тщательностью отделить абстрактные свойства структур данных, позволяющие характеризовать их внешним образом по отношению к задачам физического представления. В гл. VI изучается принципиальное и зачастую недостаточно хорошо выявленное понятие рекурсии; здесь показаны важность рекурсивных алгоритмов и методы их реализации. В гл. VII принципы и техника, развитые в предыдущих главах, применяются к построению эффективных алгоритмов в трех важных областях: при работе с массивами, сортировке и обработке текстов. Наконец, в гл. VIII содержится попытка синтезировать все методы, указанные в книге, и приводятся современные размышления о трудных проблемах методологии программирования.

Эта книга в принципе предназначена читателям, уже имеющим некоторый опыт в информатике, в том числе (хотя это и необязательно) и в одном из используемых здесь языков; и все же она была задумана как совершенно независимое целое в том

смысле, что все используемые понятия в ней же и определены. Таким образом, и новичок сможет извлечь из книги пользу, если он согласится с временными «пробелами» в изложении некоторых понятий, строгое определение которых дается после их первого использования. Книга может также служить пособием по языкам программирования, алгоритмам, методологии программирования и т.д.

Особое внимание мы уделяем терминологии. По–видимому, информатика больше, чем какая–либо другая наука, перегружена американской терминологией. За исключением нескольких официально принятых терминов, в словаре информатики царит полная анархия; примером, достаточно ясно иллюстрирующим жаргон, употребляемый некоторыми специалистами по информатике, может служить одна брошюра, выпущенная крупной фирмой–производителем ЭВМ, в которой английские «locks» и «interlocks» соседствуют с французскими «espaces–adresses» и с франко–английским «systeme processor».

Мы надеемся, что нам удалось почти везде избежать лингвистической англомании, более или менее признанной в среде специалистов в области информатики (в приложении Б можно найти словарь основных терминов английского происхождения), так что ни «информатика», ни «вычислительная машина» не будут раболепными переводами английских слов[1].

Все содержащиеся в этой книге программы, кроме коротких демонстрационных примеров, были проверены на машине. Мы старались соблюдать правила каждого языка, которым пользовались; одно исключение из этого принципа – в ФОРТРАНе – будет в соответствующем месте отмечено.

Изложение всякой научной или технической дисциплины содержит трудные (и даже сухие) разделы, и настоящая книга, возможно, не избежала этого правила. Мы надеемся, однако, что идеи не затерялись в чащобах техники. Особенно же мы рассчитываем не только сообщить читателям навыки программирования, но также (и даже прежде всего) дать ему возможность ощутить изящное в программировании и привить увлеченность этой наукой.

апрель 1978

Бертран Мейер
Клод Бодуэн

[1] Разделяя озабоченность авторов терминологической анархией, мы тем не менее кое–где отходим от несколько подчеркнутой авторами ориентации на французскую терминологию, употребляя в переводе прежде всего наиболее сложившиеся и благозвучные термины независимо от их происхождения.– Прим. *перев. и ред.*

О библиографии и упражнениях

Библиографические ссылки в квадратных скобках, такие, как [Люка 73], относятся к библиографии в конце книги. Они встречаются в тексте и, кроме того, в конце каждой главы. Библиография не претендует ни на беспристрастность, ни на полноту; приводимые ссылки есть результат выбора, по необходимости субъективного. Кроме того, мы не пытались систематически давать ссылки на хронологически первые публикации результатов, алгоритмов и т.д., когда более поздние работы оказывались легче понимаемыми или более доступными[1].

Каждая глава, за исключением последней, содержит некоторое число упражнений и задач; в конце книги приводятся решения, некоторые только в виде набросков, другие более подробно.

Трудно поблагодарить всех тех, чья помощь и влияние были для нас определяющими, и при этом не связать с ними имеющиеся неудачи и промахи. Мы хотели бы все же, уточнив, что все недостатки этой книги – наши, засвидетельствовать нашу признательность коллегам, которые нам помогли, прочитав рукопись и сделав полезные замечания, и всем тем, чьи идеи мы заимствовали в ходе многочисленных плодотворных дискуссий. Это Ж.–Р. Абриаль, А. Отэн, А. Боссави, Г. Бриссон, П. Казо, М. Демюин, М. Достатни, Г. Гайа, М. Галинье, Ж.П. Грегуар, П. Грессей, Л. Яфиль, И. Кербар, Д. Лангле, Б. Л. Оже, У. Люка, А. А. Мейер, П. Мулен, Ж. Рандон, Ж. П. Руи, Д. Спон, Ж. М. Триллон и М. К. Вилларем; особенно мы благодарим Л. Болье и Д. Тибо за все полезное, что мы извлекли из совместной подготовки обзорного курса по «общей информатике» в Эколь Насиональ Сюперьор де Телекомюникасьон и К. Кэзера за такую же работу в Институте Информатики Предприятия. Некоторые фрагменты этой книги были использованы для преподавания в этих двух вузах и в фирме Электричество Франции; мы благодарим других преподавателей и слушателей этих курсов за сотрудничество (особенно Д. Жало из ЭНСТ, который обнаружил ошибку в гл. VII). Наконец, мы выражаем свою глубокую признательность Управлению научных исследований фирмы Электричество Франции и Сервис ИМА, которые создали обстановку, позволившую подготовить эту книгу, а также Р. В. Флойду, Д. Е. Кнуту и Д. Маккарти, чьи книги дали решающий импульс всей нашей работе.

[1] Там, где это было уместно, при переводе были добавлены ссылки на аналогичную литературу на русском *языке. – Прим. ред.*

ГЛАВА I.
ОБЩИЕ СВЕДЕНИЯ О ПРОГРАММИРОВАНИИ

- Когда я беру слово, оно означает то, что я хочу сказать, не больше и не меньше, – сказал Шалтай высокомерно.

- Вопрос в том, подчинится ли оно вам, – сказала Алиса.

- Вопрос в том, кто из нас здесь хозяин – сказал Шалтай–Болтай . – Вот в чем вопрос!

Льюис Кэрролл

Алиса в Зазеркалье[1]

ОБЩИЕ СВЕДЕНИЯ О ПРОГРАММИРОВАНИИ

В этой главе изложены основные понятия, необходимые для понимания программирования: информация, информатика, вычислительная машина. Это будет, разумеется, обращением к широкому кругу читателей. Техническая часть вычислительных машин рассмотрена бегло, обсуждены только те их свойства, которые интересны программисту; внимание акцентируется на природе тех задач, которые решаются с помощью машин. Два последних раздела посвящены определению некоторых терминов, раскрывающих важные понятия, которые используются на протяжении всей книги, и историческому обзору развития информатики.

[1] Перевод Н. М. Демуровой.

I.1. Введение

У большинства наших современников всякий намек на «информатику» или «вычислительную машину» вызывает инстинктивную, из глубины души возникающую реакцию, в которой обнаруживается серьезная озабоченность. Перечень наиболее широко распространенных суждений по поводу информатики[1] можно было бы свести к нескольким основным темам: тема неизбежного господства машины над человеком (*«с этими машинами мы скоро будем не нужны»*); тема картотек и ведомостей (*«мы стали просто номерами»*); тема ошибки (*«с тех пор, как платежи идут через машину, все мои счета неверны»*); тема Голема или «ученика чародея», порождающего события, ход которых он не может остановить (*«робот покорит человека!»*); тема всемогущего предвидения (*«всего х франков за ваш гороскоп, составленный вычислительной машиной!» – «наша ЭВМ предпочитает Красавца на скачках в ближайшее воскресенье»*).

Цель этой главы совсем не в том, чтобы дать ответ на приведенные выше суждения или опасения, которые они выражают. Мы хотели бы показать, с одной стороны, возможности, открываемые информатикой, а с другой – ее пределы с той достоверностью, которая убедила бы читателя в том, что вычислительные машины не заслуживают чаще всего *«ни этих почестей, ни этого презренья»,* как сказал Расин; что они представляют собой хотя и мощные, но в то же время очень примитивные орудия; что их видимая сложность позволяет слишком часто приписывать им успехи и неудачи, за которые следовало бы считать ответственным человека, их использующего и интерпретирующего результаты их операций.

I.2. Что такое информатика?

Марсианин, прилетевший к нам и заинтересовавшийся значением слова «информатика», обратился бы, возможно, сначала к энциклопедическому словарю «Пти Робер». Несомненно, он был бы огорчен, не обнаружив там статьи «информатика»; он мог бы только увидеть (издание 1972 г.) в одном из подразделов статьи «информация» следующее упоминание:

Теория, обработка информации («Информатика»)

Принимая во внимание кавычки, которые окружают слово «информатика», наш марсианин сказал бы себе наверняка, что тут речь идет о каком–то тонком и немного туманном понятии, пугающем даже составителей словарей. Может быть, он обратился бы тогда к «общественному мнению» и получил бы, вероятно, два типа определений:

«Автоматическая обработка информации» и «наука о вычислительных машинах»

Даже придя к компромиссу типа «наука об обработке информации с помощью вычислительных машин», он вынужден был бы определять два новых понятия: «информация» и «вычислительная машина».

> Двузначность этого определения есть следствие более глубокого дуализма, который был характерной чертой информатики с самого ее начала. Говорить об информации – это значит акцентировать внимание на человеческих аспектах информатики; сказать *«наука о вычислительных машинах»* – значит настаивать на важности машины. В течение тридцати лет с тех пор, как программируют на ЭВМ, продолжает оставаться открытым вопрос: человек ли должен приспосабливаться к машине или наоборот? Другими словами, как объяснял Шалтай–Болтай: «вопрос в том, кто из нас здесь хозяин, вот в чем вопрос».

Заинтересованный двусмысленным характером слова «информатика» – слова

[1] Естественно, что этот набор примеров отражает французские реалии повседневной жизни.– *Прим. ред.*

французского происхождения, встречающегося и в немецком, и в русском[1], – наш марсианин мог бы обратиться и к другим языкам. Он констатировал бы, что вообще существуют два соперничающих выражения: одно из них, принятое в основном в университетах, означает науку о вычислительных машинах (Computer Science, Computerwissenschaft, Science des ordinateuers); другое, употребляемое в инженерной среде, – обработка информации; Electronic Data Processing (EDP), Datenverarbeitung. Это балансирование между двумя полюсами – машиной и человеком –вскрывает принципиальную двойственность.

Что можно извлечь из этих семантических колебаний? Прежде всего, что для человека, желающего понять, что такое информатика, эти два слова – «информация» и «ЭВМ» – являются, конечно, ключевыми словами; два следующих раздела соответственно посвящены изучению этих понятий. Далее, что главная проблема информатики состоит в необходимости перейти от *человеческих* понятий к *механическому* представлению. *Методы программирования,* развиваемые в этой книге, в некотором смысле есть не что иное, как средства, предназначенные для облегчения этого разъяснения.

I.3. Что такое информация?

Будем называть *информацией* всякий *критерий выбора* среди элементов некоторого множества, т.е. всякий критерий, позволяющий сократить множество, в котором разыскивается ответ на некоторый конкретный вопрос. Если, например, задача ставится так: «Какое слово самое длинное во французском языке?», то полезной информацией будут высказывания: «длина разыскиваемого слова больше 5» и «разыскиваемое слово является наречием». Эти сведения позволяют, действительно, сократить поиск сначала до множества слов длиной больше 5, затем до подмножества предыдущего множества, сформированного исключительно из наречий (т.е. множества наречий длиной больше 5). На этом этапе сведения о том, что разыскиваемое слово не является глаголом, не несут «информацию» в собственном смысле этого слова (или несут нулевое «количество информации»), так как это не позволяет еще раз сократить множество поиска – никакое французское слово не является одновременно глаголом и наречием.

> Заметим попутно, что поставленная задача сформулирована недостаточно ясно, чтобы допускать автоматическую обработку: для такой обработки нужно было бы уточнить исходное множество, «словарь», т.е. определить, что такое «слово французского языка».

Особенность информатики состоит в том, что ее методы позволяют обрабатывать очень большие словари, из которых надо будет извлекать небольшое количество результатов. «Обработка информации» – это в основном *сокращение количества информации.*

> Таким образом, перевод с французского на английский, «автоматический» или нет, требует, чтобы игнорировалась информация, содержащаяся в некоторых орфографических, синтаксических, даже семантических (нюансах, синонимах) характеристиках исходного языка, чтобы сохранить только то, что непосредственно полезно для поиска «эквивалента» в результирующем языке. В других областях вычисление среднего или максимума (например, максимальной температуры в течение месяца) порождает переход от множества с большим числом элементов (например, множества температур, замеренных в течение данного месяца) к множеству из одного элемента (максимум или среднее).

Обработка информации в том виде, в котором она была только что рассмотрена, есть практическая реализация функции f сокращения информации, функции, отображающей (рис. 1.1) множество D, называемое множеством *данных,* в множество

[1] Говоря о многозначности слова «информатика» в русском языке, следует отметить еще одно (при этом более раннее) его употребление в области, связанной с документалистикой и информационно–поисковыми системами. (См., например, Михайлов А. И., Черный А. И. Основы информатики. –М.: Наука, 1968.) – *Прим. ред.*

R, называемое множеством возможных *результатов,* таким образом, что каждому данному d, принадлежащему D, функция f ставит в соответствие результат r = f(d), принадлежащий R.

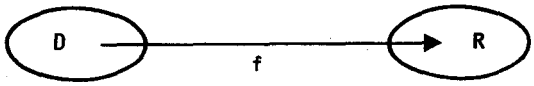


Рис. I.1 Обработка информации.

Вообще говоря, более соответствует реальности рассмотрение D как множества, называемого в математике «декартовым произведением», т.е. каждое «данное» в действительности составляется объединением нескольких элементарных данных (их «соединением» в том смысле, в котором этот термин будет употребляться в гл. V). Так, множество температур, замеренных каждый час в течение дня в октябре 1977 г., составит одно данное для функции, вычисляющей месячный максимум температур.

С практической точки зрения важно помнить, что обработка информации, автоматическая или нет, никогда не *добавляет* информацию, она состоит в том, что *извлекает* интересную информацию из той, которая содержится в данных.

Определения, которые здесь приведены, могут применяться как к ручной, так и к машинной обработке информации. И действительно, природа операций, выполняемых в обоих случаях, различается не принципиально; то, что их различает по существу, это *порядок величины* тех количеств информации, которые могут быть обработаны, т.е. допустимые размеры для множеств D и R, и скорость, с которой выполняются эти виды обработки. Теперь, для того чтобы продвинуться в нашей попытке определения информации, необходимо рассмотреть машины, осуществляющие автоматическую обработку информации, – ЭВМ.

I.4. Что такое вычислительная машина?

I.4.1. Общие положения

Тот же словарь «Пти Робер» определяет ЭВМ как «электронный вычислитель, снабженный памятью и сверхскоростными средствами вычислений, умеющий адаптировать собственную программу к условиям работы и принимать сложные решения».

В этом определении можно было бы придраться к таким деталям, как использование эпитета «электронный», который характеризует скорее применяемую здесь технику, чем действительно фундаментальные концепции, или к слову «вычислитель», которое неудачно пытается ограничить область применений исключительно числовой информацией. Мы предпочтем все же отдать должное уважение этому определению, очень характерному для концепции, согласно которой вычислительная машина «сверхбыстродействующая», имеет «большой объем памяти», может принимать «сложные решения» (очень нечеткие эпитеты для определения!), хотя не очень хорошо известно, ни что такое эти «вычисления» и эти «решения», ни в каких единицах измерены их «скорость» и их «сложность».

Чтобы выполнить обработку информации, надо, как мы видим, осуществить применение рассматриваемой функции f к некоторому данному (возможно, сложному), которое принадлежит множеству D.

f – произвольная функция, которую надо «вычислить»: перевод с французского на английский, нахождение среднего или максимума, вычисление параметров железобетонного моста, преобразование символьных выражений, определение лучшего скакуна на скачках и т.д.

Чтобы «промоделировать» f, необходимо располагать тремя *физическими представлениями* (Рис. I.2):

- D', физическим представлением множества данных D;

- R', физическим представлением множества результатов R;
- f, физическим представлением функции обработки f, которое должно быть преобразованием или последовательностью преобразований, оперирующих над D' и дающих в результате элемент R'; Г должна быть практически *реализуемой,* т.е. должна «вычисляться» на некотором физическом устройстве.

Чтобы выполнить обработку информации и интерпретировать ее результат, необходимо располагать, кроме того, *двумя кодами представлений* φ и ψ (Рис. I.2):

- φ позволяет *представить* данное из D (абстрактное) с помощью элемента из D' (физического): это *входной код;*
- ψ позволяет *интерпретировать* элемент из R', результат автоматической обработки, в качестве представления результата, принадлежащего R: это *выходной код.*

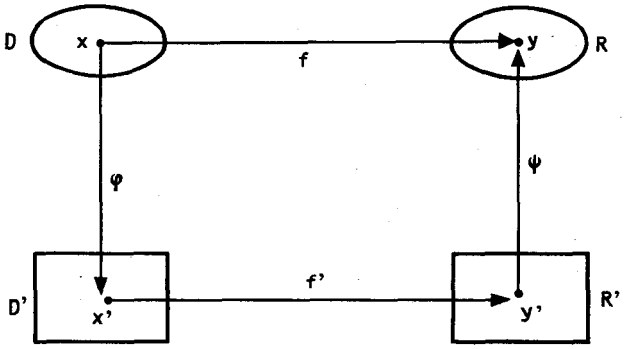


Рис. I.2 Обработка информации на вычислительной машине.

Программа обработки информации, реализуемой функцией f, является, таким образом, данным, составленным из элементов, физически представимых (конструируемых) множеств D' и R', кодов φ и ψ и автоматизируемого преобразования f ', таких, что для всякого данного x (принадлежащего D) имеет место

$$f(x) = \psi\ (f'(\varphi\ (x))).$$

ЭВМ позволяет получить такие физические представления. Принципиально машина состоит из двух частей (Рис. I.2):

- *центрального процессора* – совокупности электронных (иногда электромеханических) устройств, позволяющих «выполнить» f ';

- *памяти,* физической системы, которая в зависимости от ситуации будет интерпретироваться как D' или R'.

Последнее положение требует некоторых пояснений. Память – это средство представления «информации», т.е., как было показано, элементов из некоторых множеств; Память вычислительной машины есть физическая система с большим числом состояний; каждое состояние может быть связано (в этом состоит роль φ и ψ) с элементом какого–либо множества.

Практически используемые системы, т.е. системы, в которых можно эффективно различать их состояния, имеют только *конечное число состояний;* иными словами, множества D и R всегда представимы **конечными множествами.** Это одна из фундаментальных черт информатики: рассматриваемые множества конечны; когда говорят о бесконечных множествах, речь идет о некоторой удобной для рассуждений идеальной конструкции; эти множества будут аппроксимироваться иногда «большими» подмножествами, но всегда конечными.

Рис. I.3 Вычислительная машина.

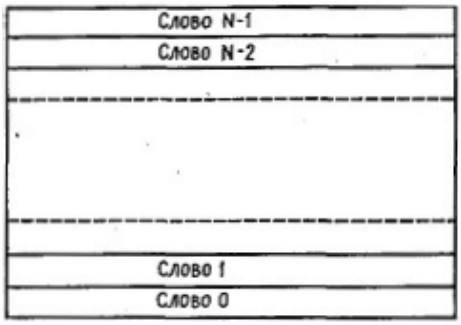


Рис. I.4 Память (так называемая модель фон Неймана).

Удобно представлять память как множество из N физических систем с M состояниями (лучше, чем представление ее единой системой с M^N состояниями); каждая из таких систем называется **словом,** и память организуется из множества N слов (Рис. I.4).

В современной технике обычными для M являются значения от $2^8 = 256$ до 2^{60} (примерно 10^{18}), а для N – начиная с нескольких тысяч.

При помощи такой схемы одна и та же память может быть интерпретирована и как D', и как R' не только для одной конкретной задачи, а для любой из задач, лишь бы число элементов множеств, которые мы хотим представить (D и R), не было слишком большим (меньше или равно M^N) и были уточнены коды представлений φ и ψ. Одно из возможных состояний одного и того же *слова* может интерпретироваться в зависимости от ситуации и кодов как:

- целое натуральное число между 0 и M–1 включительно;
- целое положительное, отрицательное или нуль между – m и m включительно ($2m + 1 < M$);
- слово[1] из p букв обычного алфавита[2], если $26^p \le M$,
- и т.д.

Как функционирует второе устройство на Рис. I.2 – центральный процессор, которому поручено выполнение операций, составляющих f'? Конкретная вычислительная машина должна иметь конечное число физически реализованных, «встроенных», базовых выполняемых операций: одна из главных идей создателей информатики состояла, таким образом, в том, что с помощью некоторого кода можно было указать соответствие между множеством возможных операций центрального процессора и словом (или частью слова). Это и составляет принцип *вычислительной машины с хранимой программой,* по которому элементы памяти могут содержать информацию, представляющую в равной степени данные (элементы из D), результаты (элементы из R), а также *операторы,* выбранные среди операций, которые может выполнять машина[3]. В отличие от предшествовавших специализированных вычислительных машин (как, например, суммирующая машина Паскаля) ЭВМ является *универсальной* машиной, способной выполнять все виды преобразований f, которые представляются как последовательность f' операций, выполняемых обрабатывающим устройством. Одна и та же ЭВМ становится, таким образом, поочередно или даже одновременно переводчиком, математиком, инженером: для этого достаточно (мы еще часто будем употреблять это выражение – «для этого достаточно!») уметь выражать представляемые функции в терминах элементарных выполняемых преобразований.

[1] «Слово» имеет здесь свой обычный смысл, а не «информатический», определенный несколько выше.

[2] Французский алфавит насчитывает 26 букв; для русского читателя обычным можно было бы считать алфавит из 32 букв, и тогда неравенство этой фразы должно было бы иметь вид $32^p \le M$ – *Прим. перев.*

[3] К списку сведений, которые может содержать память, следовало бы добавить информацию, непосредственно используемую программой в ходе ее выполнения.

Такое разбиение на элементарные задачи называется алгоритмом или программой; мы будем рассматривать эти два термина как синонимы, хотя первый из них обозначает обычно действие более абстрактное, чем второй.

Понятие машины с хранимой в памяти программой приводит к уточнению нашей схемы ЭВМ (Рис. I.5).

Память содержит теперь «данные», «результаты», а также операторы; что касается центрального процессора, он больше не ограничивается выполнением «вычислений» в строгом смысле, т.е. выполнением базовых операций, входящих в f', но должен также управлять последовательностью этих операций во времени: речь идет О том, чтобы определить после выполнения каждого оператора, какой оператор выполняется следующим. Таким образом, центральный процессор логически состоит из двух частей:

- **Управляющее устройство** определяет последовательность операторов во времени и занимается поиском в памяти выполняемых операторов. Оно имеет, как это изображено на рисунке, *регистр команд,* в котором располагается оператор (команда), подлежащий выполнению.

- **Обрабатывающее,** или **арифметическое, устройство** выполняет операторы, передаваемые управляющим устройством; к нему относится показанный на рисунке *общий регистр,* который получает операнды и результаты этих операторов. Современные машины имеют от одного до нескольких десятков регистров.

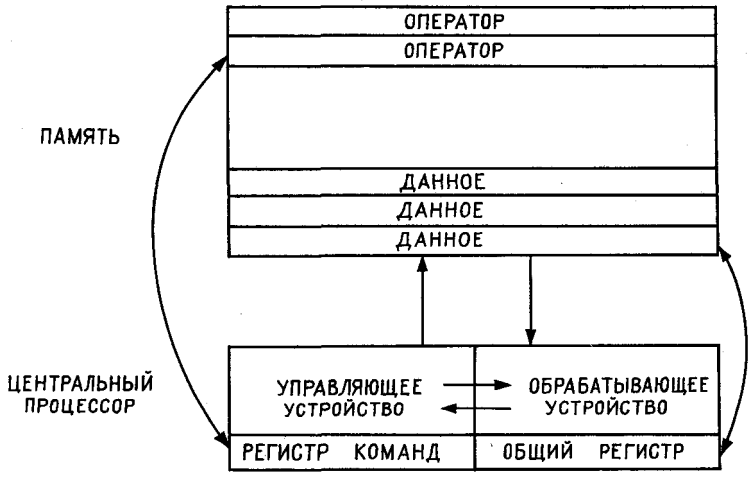


Рис. I.5 Вычислительная машина (так называемая модель фон Неймана).

I.4.2. Подробнее о памяти

До сих пор очень кратко было сказано о том, что такое «физические системы с конечным числом состояний», которые образуют *слова* памяти. Теперь пора приоткрыть край занавеса.

В силу очевидного естественного закона, о котором можно было бы долго рассуждать, наиболее легко используемыми физическими системами являются почти всегда *двоичные,* т.е. элементы с двумя возможными состояниями. Такой элемент называется «бит» (*binary digit,* двоичная цифра). Безразлично, как называть два этих состояния: 0 и 1, + и –, истина и ложь, Пат и Паташон[1]...

Разумеется, для того чтобы *слово* могло изображать используемую информацию, оно должно иметь более двух возможных состояний. Оно становится, таким образом, *комбинацией битов:* действительно, система, образованная из n элементов, где у каждого элемента – два состояния, сама имеет 2^n возможных состояний.

[1] В оригинале – Лорель и Арди, два очень известных на Западе кинокомика, весьма заметно различающихся своими габаритами. – *Прим. перев.*

Размер слова, n, является характеристикой каждой ЭВМ и изменяется в достаточно широких пределах при современном состоянии техники. Наиболее обычные значения n = 8,16,24,32,48,60,64,....

В силу многообразия возможных размеров слов для измерения «объемов памяти» часто прибегают к более мелким единицам измерения: байт, множество из 8 битов (система с $2^8 = 256$ состояниями) или **литера,** единица более расплывчатая (6, 7 или 8 битов). Отметим попутно обозначения «К–байт», «К–слов» и т.д. для обозначения размеров памяти, где «К» означает «кило» (тысячу), но «тысячу» несколько особую, $2^{10} = 1024$: степени двойки удобны для некоторых операций.

С точки зрения физической конструкции элементы, входящие в состав систем представления информации – информационные носители, относятся к двум категориям:

- **механические** системы, примером которых может служить перфокарта: на пересечении конкретной строчки и конкретной колонки карты находится элемент в двух возможных состояниях, состояние «проперфорированное» и состояние «непроперфорированное». Классическая **перфокарта** есть множество из 80 колонок, в каждой из которых 12 элементов такого типа. *Перфоленты* подчиняются аналогичному принципу;
- **электронные** системы, состоящие из элементов намагничивающей среды, каждый из которых имеет два состояния, «намагниченное» и «размагниченное» (*ферритовые сердечники,* элементы дорожки *магнитной ленты,* дорожки *магнитного диска*), или из *транзисторов* с двумя состояниями, «проводящим» и «непроводящим», или из других типов «полупроводников» и т.д.

В памяти (запоминающих устройствах) современных ЭВМ используются только элементы второй категории (они заменили предшествовавшие электромеханические системы). Механические элементы применяются для хранения информации вне машины и для переноса такой информации в память. Этой проблемой мы сейчас и займемся.

I.4.3. Ввод–вывод

Если вернуться теперь к нашей логической схеме обработки информации (Рис. I.6), то способ, которым вычислительная машина поможет нам представить D', R' и f', должен показаться более ясным. Чтобы успешно выполнить вычисления f, мы должны еще уточнить, что такое φ – входной код и ψ – выходной код. Другими словами, остается уточнить, как можно сообщить машине способ преобразования внешней формы данных (D) во внутреннюю (D'), затем внутренней формы результатов (R') во внешнюю форму (R). Под «внутренней» и «внешней» формами здесь подразумеваются «машинная» и «человеческая» соответственно.

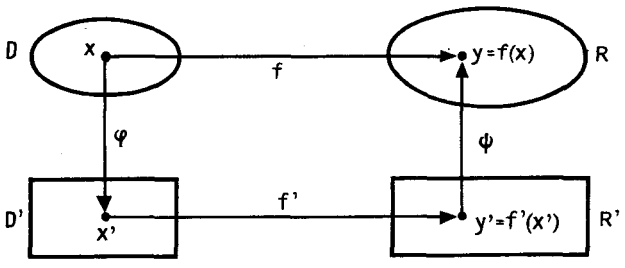


Рис. I.6 Обработка информации.

Это преобразование выполняется с помощью устройств, называемых **периферийными,** которые обеспечивают общение машины с внешним миром. Различают

- устройства **ввода** или **чтения,** которые позволяют передать информацию в память машины. Примеры: устройство чтения с перфокарт или перфолент, лентопротяжный механизм, магнитный диск, клавиатура подключенной к ЭВМ пишущей машинки, оптическое читающее устройство, устройство чтения магнитных карт и т.д.;

- устройства **вывода** или **воспроизведения (записи),** которые позволяют вычислительной машине передать информацию из памяти во вне (этим внешним получателем информации может быть либо человек, который хочет получить результаты обработки в понятной ему форме, либо другая машина). Примеры: печатающее устройство, выходной ленточный или карточный перфоратор, магнитная лента или магнитный диск, телетайп, экран терминала, перо графопостроителя, органы управления программно–управляемым станком и т.д.

Заметим, что некоторые периферийные устройства, такие, как диски или ленты, могут с равным успехом использоваться как для чтения, так и для записи.

На Рис. I.7 мы дополнили схему вычислительной машины периферийными органами, связанными с ЭВМ устройствами обмена, которые могут обслуживать сразу несколько периферийных устройств.

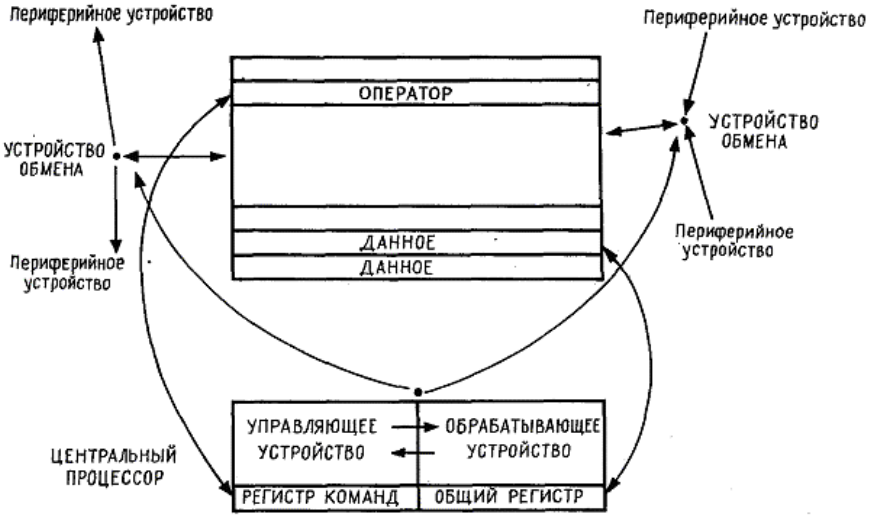


Рис. I.7 Вычислительная машина

I.4.4. Оперативная память, внешняя память

В рассматривавшейся до сих пор классической модели вычислительной машины, называемой моделью фон Неймана[1], память образована из N элементов, или *слов,* пронумерованных от 0 до N–1; «номер» слова называется его адресом. В этой модели все слова эквивалентны в том смысле, что время доступа к слову (время, необходимое для его передачи из памяти в общий регистр или в регистр команд) практически не зависит от адреса (для определенности, это время может иметь порядок миллионной доли секунды на современных ЭВМ).

Этот тип памяти (для которого техника использует обычно ферритовые сердечники или полупроводники) практичен, но дорог. Поэтому ее предназначают для «ядра» вычислительной машины, в котором она составляет главную, или оперативную память.

В современных вычислительных системах объем оперативной памяти варьируется от 8К–байтов на малых машинах до нескольких тысяч К–байтов на больших системах (напомним, что 1 байт это набор из 8 бит и что К = 1024). Если

[1] Джон фон Нейман (1903–1957), математик, логик и физик, родившийся в Венгрии. Существуют «модели» более развитые, чем «машина фон Неймана». Они отличаются, в частности, доступом к памяти, которая рассматривается необязательно как однородное множество совершенно одинаковых слов.

решение одной и тем более нескольких задач требует больших объемов запоминающих устройств, чем размер оперативной памяти, необходимо обращаться к **внешней памяти,** которая имеет большую емкость, но в то же время характеризуется двумя следующими свойствами:

а) время доступа, как правило, существенно больше;

б) главное, это время не постоянно: время, необходимое для перемещения слова, зависит от места, из которого выполнялось предыдущее перемещение.

Носителями внешней памяти могут быть магнитные диски, магнитные барабаны, магнитные ленты и т.д. Действительно, можно рассматривать входы и выходы предыдущего раздела как общение между оперативной и внешней памятью.

В заключение заметим, что по существу и независимо от проблем быстродействия имеются два типа памяти:

– *последовательная память,* образованная последовательностью «зон», где возможны только две операции:

- выбрать следующую зону[1],
- вернуться к началу.

Примером этого типа памяти является магнитная лента.

– *память с произвольным доступом,* где можно обратиться к любой записи[2], если она отмечена *ключом,* который в простейшем случае является попросту номером или адресом (оперативная память на ферритовых сердечниках, диски, барабаны).

I.4.5. Порядок некоторых величин

В таблице Рис. I.8 даны порядки величин быстродействия различных устройств. В качестве единицы обмена информации принята одна литера.

Еще раз подчеркиваем, что речь идет о порядке величин, а не о точных значениях; точные числа в данном случае значат меньше, чем их относительные величины.

Больше всего бросается в глаза в табл. Рис. I.8 наличие пяти четко различающихся уровней быстродействия, размещенных в порядке от самых быстрых к наиболее медленным:

- выполнение оператора (машинной команды): несколько сотен наносекунд, т.е. несколько десятимиллионных долей секунды;
- *чисто электронные* пересылки между оперативной памятью и центральным процессором: порядка *миллионной доли секунды.* Заметим, что, несмотря на более высокую скорость выполнения каждой команды, быстродействие этих пересылок влияет определяющим образом на подсчет количества операций, выполняемых в секунду: не надо забывать, что до того, как выполнить каждую команду, ее надо взять из оперативной памяти;
- пересылки, заставляющие работать *магнитные* носители с *механическими* перемещениями, такие, как диск или лента. В этом случае пересылка в собственном смысле этого слова производится с «магнитными» скоростями (порядка миллионной доли секунды), но предварительно необходимо должным образом расположить носитель по отношению к читающей головке, а это может потребовать разного времени, от нескольких

[1] Или в некоторых случаях предыдущую зону. См. также ниже в V.6.2 «функциональную спецификацию» линейных списков.

[2] Следует отметить русскую омонимию, неизбежно вносимую при переводе: термин «запись» в отечественной литературе по вычислительной технике и информатике обозначает либо процесс регистрации информации, занесения ее на носители (французское ecriture), либо, определенным образом организованную порцию информации, чаще всего являющуюся единицей обмена между оперативной и внешней памятями (французское enregistrement). Всюду, где читателю не составит труда различать эти омонимичные значения, в переводе сохранена принятая русская терминология. – *Прим. перев.*

миллисекунд (приблизительно 30 мс для диска) до многих десятков секунд (полная перемотка ленты). Порядок, в котором организуется доступ к элементам, является здесь существенным;

- пересылки *электромеханические:* читающий перфоратор, алфавитно–цифровое печатающее устройство; скорости порядка *тысячной доли секунды* на литеру, т.е. значительно ниже, чем на предыдущем уровне;
- наконец, пересылки, включающие безнадежную медлительность человеческой психомоторной деятельности: порядка 10 литер в секунду для самых способных перфораторщиц.

Чаще всего было бы предпочтительно не знать эти физические ограничения и считать, что существует единая память большого размера с переменным временем доступа. И если мы все же отмечаем эти величины, то только потому, что различие их весьма существенно. Например, отношение между временем доступа к некоторому данному в оперативной памяти (примерно 10^{-6} с) и временем доступа к непосредственно следующему уровню, диску (50×10^{-3}с. с учетом среднего времени подвода) равно примерно 50000; можно сравнить программиста с ремесленником, который располагает полкой, вмещающей 20 инструментов, в метре от себя и складом, способным разместить 1000 инструментов, но расположенным в 50 километрах! Ясно, что в таких условиях выбор рядом хранящихся элементов является решающим – даже если системы, называемые в программировании *виртуальной памятью* (I.7), освобождают частично сегодняшнего программиста от этого бремени.

Мы не будем подробно изучать проблему распределения объектов между оперативной и внешней памятью; проблема эта, однако, практически очень важна, даже если надеяться, что технический прогресс позволит когда–нибудь от нее освободиться.

Арифметическая операция			100 наносекунд
пересылка	Управляющее устройство	⇔ Память	1 микросекунда
пересылка	Переферийные устройства	⇔ Память	
	Диск		1 микросекунда + подвод (до 50 миллисекунд)
	Лента		10 микросекунд + подвод (до нескольких секунд)
	Печатающее устройство		0,5 миллисекунд
	Читающее карты устройство		1 миллисекунда
	Карточный перфоратор автоматический		3 миллисекунды
	Телетайп		0,1 секунды
Ручная перфорация карт			0,1 секунды

1наносекунда = 1 миллиардная секунды
1микросекунды = 1 миллионная секунды
1миллисекунда = 1 тысячная секунды

Рис. I.8 Порядок некоторых величин.

I.5. Что может делать вычислительная машина?

Чтобы дополнить наше представление об ЭВМ и ее реальных возможностях, остается уточнить (немного) природу пресловутых «базовых операций» центрального процессора, тех, из которых, как говорилось выше, можно составлять программы.

Можно подразделить эти операции, называемые командами или операторами «машинного языка», на четыре большие категории, которые следуют, впрочем, из структуры ЭВМ.

Ниже предполагается, что каждая команда описывается постоянной частью, «кодом», и переменной, «адресной» частью. Различают:

а) *Вычислительные* команды, или команды *обработки,* выполняемые арифметическим устройством над величинами, обычно содержащимися в общих регистрах: сложение, логическое И, исключающее ИЛИ и т.д. (Рис. I.9).

б) Команды *обмена* информацией между памятью и арифметическим устройством: загрузка в регистр содержимого некоторого слова (номер регистра и адрес слова указываются в переменной части команды) или наоборот и т.д. (Рис. I.10).

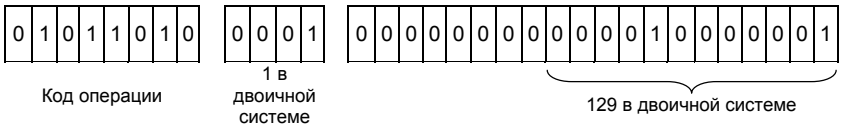


Рис. I.9 Сложить содержимое регистра 1 и содержимое слова по адресу 129; Результат в регистре 1.

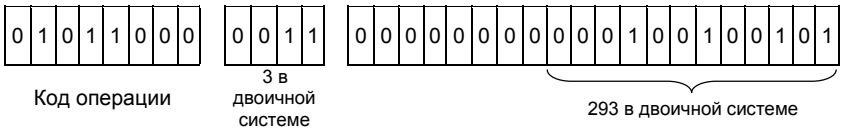


Рис. I.10 Заслать в регистр 3 содержимое слова по адресу 293.

в) Команды *проверки.* Обычно команды выполняются одна за другой в порядке адресов программы; команда проверки позволяет указать, что если проверено некоторое условие, то следующая команда вопреки обычному порядку должна быть разыскана по адресу, заданному в адресной части команды. «Условие» может быть, например, равенством содержимого двух регистров (Рис. I.11).

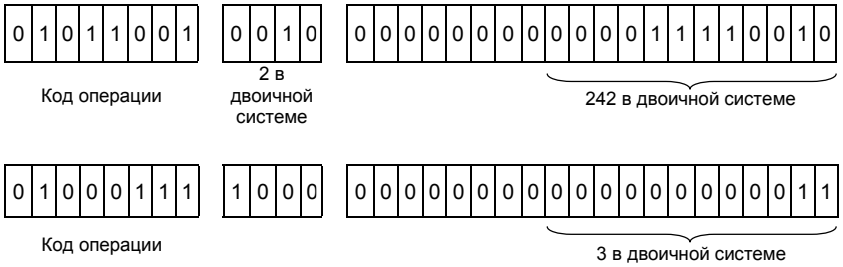


Рис. I.11 «Передать управление» команде, являющейся значением слова по адресу 3, если содержимое регистра 2 равно слову по аресу 242; иначе перейти к следующей команде (2 команды).

г) Команды *ввода* и *вывода,* задающие обмен информацией между оперативной памятью и периферийными устройствами.

Самое поразительное в этих командах, существующих в таком виде в современных ЭВМ, это их предельная простота и, если так можно выразиться, их *предельная* слабость: речь идет в конечном итоге о примитивных манипуляциях над какими–то битами, весьма далекими от задач, решаемых на машине. Самое меньшее, что можно сказать, это констатировать, что существует пропасть между «Вычислить дату ближайшего лунного затмения, наблюдаемого в Мурманске» и «Загрузить содержимое ячейки с адресом 2324 в регистр 5». Эта дистанция – одна из больших проблем программирования, которое можно определить как искусство использовать

мощность вычислительных машин, чтобы превратить в «интеллект» предельную *ограниченность* отдельных команд.

I.6. Что такое программирование?

Фактически почти никто не программирует теперь с помощью аппаратно реализованных команд, какие мы только что рассматривали. Это обстоятельство имеет несколько причин:

а) Практические трудности кодирования: не будем забывать, что внутренний код машины – *двоичный.*

б) Очень большое расстояние, упоминавшееся уже, между человеческими понятиями (уровень решаемой проблемы) и примитивным характером команд.

в) Трудность понимания длинных программ.

г) Увеличенные возможности человеческих ошибок в программах и трудоемкость процессов поиска и исправления ошибок.

д) Тесная зависимость между программой и машиной, на которой она выполняется, зависимость, делающая почти невозможным переход на другую машину.

е) Проблема *разделения:* для современных ЭВМ характерно, что несколько программ выполняются одновременно на одной и той же машине; жизненно необходимо поэтому уметь контролировать разделение «ресурсов» (устройства обработки, памяти, периферийных устройств...) и, следовательно, например, запрещать программам свободно включать те или иные команды ввода и вывода.

ж) Проблема, тесно связанная с предыдущей, – *защита* некоторых ресурсов, например некоторых областей памяти, которые содержат важную информацию и, значит, должны быть защищены от неразрешенного доступа.

Некоторые из этих проблем, например а) или в крайнем случае е) или ж), могут быть решены применением *системы кодирования,* позволяющей программисту записывать команды символически в содержательных обозначениях. Так, вместо команды Рис. I.10 можно писать в формализме, называемом **языком ассемблирования** (в противоположность «языку» команд машины, или **машинному языку**):

CONST ANTE X = 293 CHARGER X R3

или в русской версии:

ПРИСВОИТЬ X = 293 ЗАГРУЗИТЬ X R3

Специальная программа, называемая **ассемблером**, которой оснащены почти все вычислительные машины, берет на себя перевод написанных выше строк в команду Рис. I.10.

Это не решает, однако, основных упомянутых выше проблем. Именно поэтому очень рано (с начала пятидесятых годов) были изобретены **языки программирования**, которые представляют собой системы описания программ, достаточно близкие к человеческому языку, чтобы программы можно было легко построить, понять и модифицировать, но в то же время достаточно строго определенные, чтобы те же программы могли быть в конечном итоге выполнены машиной.

Чтобы выполнить программу P, написанную на таком языке программирования L, надо сделать его понятным машине. Важно знать два метода, которые позволяют получить этот результат:

а) В методе *интерпретации* соответствие между языком и машинным языком

устанавливается благодаря специальной программе, называемой **интерпретатором** и оперирующей с программой P и данными D, которые надо передать программе P. Интерпретатор декодирует P и по мере этого декодирования выполняет соответствующие команды, применяя их к элементам из D (Рис. I.12). Заметим, что интерпретатор сам по себе является программой, которая может быть написана на машинном языке или в свою очередь быть результатом более сложной программы.

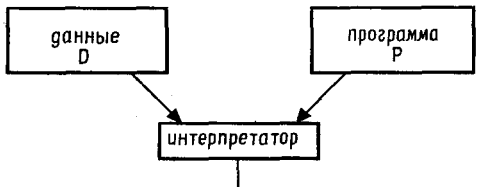


Рис. I.12 Выполнение методом интерпретации.

б) В методе *компиляции* выполнение программы включает две фазы; первая есть *перевод* программы P в эквивалентную программу F, выраженную в машинном языке; этот перевод выполняется специальной программой, которая называется **компилятором**. Второй этап – это выполнение результирующей программы P' с данными D (Рис. I.13).

Следует заметить, что решения, принимаемые на практике, часто бывают смешанными: полуинтерпретация, полукомпиляция. Понятия интерпретации и компиляции выходят за рамки проблем использования программ, написанных на языках программирования; мы вернемся к ним в гл. VII в связи с поисками эффективных алгоритмов.

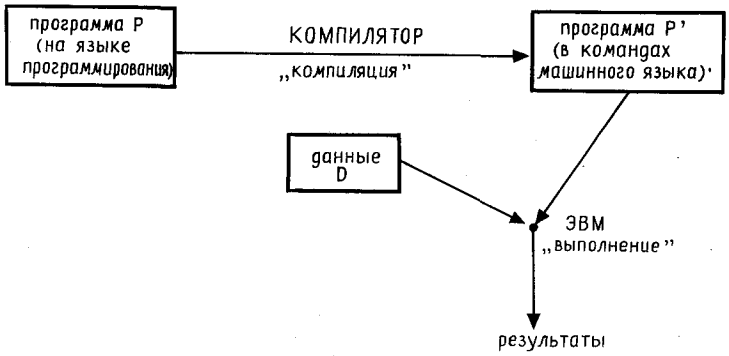


Рис. I.13 Выполнение методом компиляции.

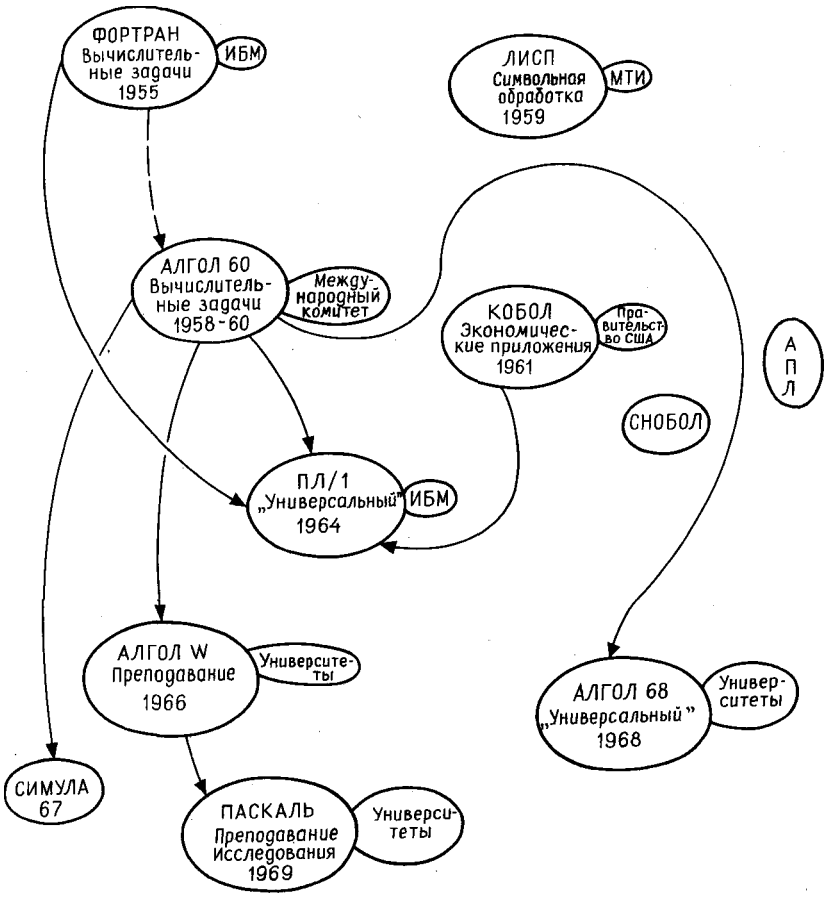


Рис. I.14 Языки программирования.

В настоящее время в мире существуют несколько сотен языков программирования. На Рис. I.14 показаны некоторые из наиболее важных языков с указанием их «родства» и влияний. Одним из первых языков программирования был ФОРТРАН (1956), который остается одним из наиболее используемых, несмотря на его достаточно примитивный стиль.

Из этого введения, мы полагаем, должно быть понятным, что информатика является очень мощным орудием, которое бросает вызов нашему интеллекту. Большая трудность этой дисциплины состоит, действительно, в том, что вычислительная машина дает средство не бояться задач таких размеров, которые превосходят все ранее рассматриваемые, а значит, такой *сложности,* к усвоению которой человеческий разум плохо подготовлен. Потребность в таком исключительно точном описании шаг за шагом последовательности преобразований информации с такой строгостью, которая не требуется ни при каком виде человеческой деятельности (даже в математике), вызвала у выдающихся программистов мысли о *трудности* программирования, требующей не столько освоения какого–нибудь конкретного языка, сколько нового способа мышления.

I.7. Несколько ключевых слов[1]

Для понимания следующих глав будут необходимы определения нескольких основных терминов программирования.

Техническое обеспечение: множество физических ресурсов (центральный

[1] Отечественную терминологию информатики нельзя признать совершенно удачной потому, что у нас уже устоялись, к сожалению, длинные словосочетания: электронная вычислительная машина (ср. англ. computer и франц. ordinateur), программное обеспечение (анг. software и франц. logiciel), техническое обеспечение (англ. hardware и франц. materiel) и т.д. Французские программисты устояли перед соблазном взять ранее родившиеся американские термины и предложили свою, краткую и даже более выразительную терминологию. – *Прим. перев.*

процессор, периферийные устройства и т.д.), составляющих вычислительный комплекс. Программисты фамильярно зовут его «железом». Употребляется в качестве противопоставления понятию *программного обеспечения.*

Программное обеспечение: множество программ, позволяющих работать с вычислительной машиной. Употребляется в качестве противопоставления понятию *технического обеспечения.*

Операционная система, или **базовое программное обеспечение:** множество программ, дополняющих функции технического обеспечения и реализующих общение между пользователями машины и ее физическими ресурсами. Можно, в частности, разрешить нескольким программам неявно использовать одну и ту же ЭВМ одновременно (или даже фактически – для некоторых машин, имеющих несколько центральных процессоров); управлять доступом к памяти, к периферийным устройствам и т.д. В широком смысле, включает ассемблеры, интерпретаторы, компиляторы.

Виртуальная память: свойство, реализованное на некоторых машинах и позволяющее программистам использовать область памяти, заведомо превосходящую размер выделенной им оперативной памяти. Часть программных объектов (данные, операторы) в каждый момент выполнения программы размещена в оперативной памяти, остальные находятся на периферийных запоминающих устройствах. С необходимостью возникают задачи обменов, но программист ими не занимается; в зависимости от каждой конкретной реализации эти задачи решаются либо средствами технического обеспечения, либо программного (операционной системой), либо комбинацией того и другого.

Файл: организованная совокупность логически связанных между собой данных, размещенных в памяти (оперативной или внешней) ЭВМ. Примеры: файл, описывающий сведения о служащих предприятия, файл библиотечных архивов.

Искусственный интеллект: исследования методов, позволяющих передать автоматическим машинам (и, в частности, ЭВМ) решение задач, традиционно рассматриваемых как достояние человеческого интеллекта: анализ естественных языков; исследование и доказательства математических теорем; «интеллектуальные» игры; манипуляции объектами в меняющейся обстановке (роботы) и т.д.

I.8. Краткая история информатики[1]

Хотя информатика как сформировавшаяся наука является молодой дисциплиной, нельзя считать неуместным разговор о ее истории, кстати восходящей к очень далеким временам, также как нельзя считать бесполезным для программиста знать ее главные вехи.

I.8.1. Прединформатика

Идея использования материальных носителей для обработки больших объемов данных, несомненно, очень стара. Кнут [Кнут 72], например, видит применение принципов автоматической обработки информации в числовых таблицах на вавилонских глиняных горшках, датируемых 2500 г. до н.э. Обычно в качестве предка современных счетных устройств упоминают также древние китайские счеты (ранее 500 г. до н.э.).

Ближе к нашему времени, в эпоху Возрождения и средние века начали появляться механические и полумеханические вычислительные устройства. Непер изобретает логарифмы в начале XVII в.; в 1620 г. появляется счетная линейка. Два

[1] Это резюме навеяно в числе других [Хаски 76].

гения своего времени предлагают оставившие заметный след в истории машины, выполняющие арифметические операции: Паскаль в 1642 г. (сложение и вычитание) и Лейбниц в 1671 г. (четыре операции). Множество различных машин появляется затем, вплоть до начала XIX в.

В VIII в. мода в европейских салонах на автоматы всех видов (среди них автоматы Вокансона, свидетельствующие об удивительной технической виртуозности, а также «шахматист» австрийца фон Кемпелена, который скрывал скорчившегося человека, были ложными отголосками эпохи) позволила подтвердить мастерство техников в области автоматических устройств.

Развитие этой техники стимулировалось зарождающейся индустрией, в частности текстильной промышленностью, для которой Жаккар, усовершенствовав в 1801 г. одно из изобретений Вокансона (1745), изобрел перфокарту, предназначенную для управления ткацкими станками.

Более века спустя англичанин Бэббидж предложил в 1822 г. свою «дифференциальную машину», а позднее, в 1833 г. «аналитическую машину», настоящую модель вычислительной машины с центральным процессором (mill, «мельница») и памятью (store, «склад»). Машина Бэббиджа осталась, однако, нереализованным замыслом, а почти все его идеи независимо «переоткрывались» в сороковые годы нашего века.

Конец XIX в. принес изобретение и усовершенствование арифмометра (Однер, Швеция, 1874; французы Томас де Кольмар с сыном, «арифмометр», 1822–1879 и Боле 1887; Барроуз, США, 1892). Это движение продолжалось до 1930 г.

Американский инженер Герман Холлерит применил для переписи населения 1890 г., а позднее усовершенствовал для следующих переписей «табулятор», использующий перфорированные карты, – машину, идеи которой восприняты алгоритмами сортировки распределением (VII.3.8).

I.8.2. Протоинформатика

В статье 1976 г. [Эккерт 76] один из пионеров американской вычислительной техники Дж. Преспер Эккерт заметил, что вычислительные машины могли бы быть изобретены на несколько десятилетий раньше – техника электронных ламп в то время уже существовала. Но только экономические и военные потребности второй мировой войны дали толчок процессу реализации.

И все же предшествующий период – время между двумя мировыми войнами – был отмечен важным этапом: установкой теоретических вех. Действительно, в 1936 г. английский математик Тьюринг в важной статье [Тьюринг 36] заложил основы математической теории вычислений и показал за несколько лет до реализации первых ЭВМ (одна из которых, британский проект ACE, был реализован под руководством Тьюринга в 1946 г.), что можно делать теоретически на вычислительной машине и что невозможно. Другие, близкие или дополняющие теории были развиты между 1936 и 1940 гг. американцами Постом и Клини и советским математиком Марковым.

I.8.3. Информатика

Дата, отделяющая протоинформатику от информатики, так же неопределенна, как дата, отмечающая переход от протоистории к новейшей истории. Несомненно, однако, что «военные усилия» во многих странах дали решающий импульс рождению первых вычислительных машин, заслуживающих этого имени:

- Ц2 и Ц3 немецкого инженера К. Цузе, готовые соответственно в 1939 и 1941 гг., но уничтоженные в конце войны;

- серия машин МАРК, построенная Г. Айкеном (Гарвардский университет и

ИБМ), первый экземпляр которой – МАРК 1 – был закончен в 1944 г.;

- ЭНИАК Джона Маучли и Дж. П. Эккерта (Пенсильванский университет, 1943–1946 гг.); эти два изобретателя основали в 1946 г. компанию «Унивак», первая машина которой вышла в 1951 г.

- ЭДСАК англичанина Уилкса (Кэмбриджский университет), строившийся с 1946 по 1949 г.

Далее следует, если мы хотим избежать скучного перечисления аббревиатур и фамилий, отметить лишь несколько важных этапов в эволюции техники и концепций:

- первое использование индексного регистра на машине Манчестерского университета (1947–48 гг.)

- изобретение транзисторов в 1948 г. исследователями компании «Белл»;

- изготовление серии ИБМ 701 начиная с 1953 г.;

- создание ФОРТРАНа в 1956 г. [Бэкус 57]; в том же году М. Перре изобрел французское слово “ordinateur”[1];

- первое описание АЛГОЛа Бэкусом в 1959 »г., пересмотренное и исправленное в [Наур 60]; здесь можно найти первую формальную спецификацию синтаксиса языка программирования;

- описание КОБОЛа в 1959 г. в результате сотрудничества американского правительства, пользователей и конструкторов;

- первая система с виртуальной памятью (Атлас, Манчестерский университет, 1962 г.).

Начиная с этого времени расширение мирового парка вычислительных машин идет с высокой и все увеличивающейся скоростью; Европа пытается сократить свое отставание в оборудовании (Рис. I.15), однако в силу мало согласованного сопротивления американскому примату, а также из–за определенной медлительности в развитии преподавания информатики в университетах и технических вузах сохраняется все–таки сильная зависимость от языков и машин, импортируемых из США.

	США	Западная Европа	Франция
1960	5400	1000	30
1965	23 000	7600	
1970	70000	32000	5000
1975	120000	70000	15000

Рис. I.15 Эволюция парка ЭВМ.

В самое последнее время в области технического обеспечения (что касается программного обеспечения ему посвящена остальная часть книги) произошел заметный прогресс миниатюризации: «микропроцессоры», процессоры с низкой стоимостью и очень малыми размерами, приближаются все больше по своим возможностям к центральным процессорам «классических» вычислительных машин; стоимость элементов оперативной памяти точно так же уменьшается. В то же время в области многочисленных информационных систем происходит эволюция к децентрализованным структурам («распределенная обработка») при этом используются связи на больших расстояниях («телеобработка»).

[1] Этим отмечается этап французской истории информатики. – *Прим. перев.*

Библиография

Существует много введений в информатику. Для беглого знакомства с проблемами этой науки можно прочитать, например, [Арсак 70].

Для более полного описания принципов программирования и фундаментальных свойств вычислительных машин можно обратиться к [Вегнер 69] или [Гир 69]. Из работ, посвященных углубленному исследованию структур ЭВМ, надо назвать [Мейнадье 71] на французском языке и [Белл 71] на английском.

Среди введений в программирование, излагающих современные идеи по этому вопросу, можно рекомендовать [Наур 74], а также [Вирт 72], ориентированные на язык программирования Паскаль. Более ранний выпуск [Дейкстра 71] прежде всего немного более труден и менее многословен, но он сыграл важную роль в развитии методов преподавания программирования[1].

ЗАДАЧА

I.1 Алфавитный порядок

Следующее упражнение на первый взгляд не имеет отношения к технике информатики, но в действительности оно хорошо показывает тот вид трудностей, с которыми встречаются в программировании (подробности в гл. VIII**).**

Считается известным алфавитный порядок букв (A B C.. .X Y Z), и *словом* называется конечная последовательность букв, относительно которых для простоты предполагается, что все они прописные и в последовательность не включены ни пробелы, ни тире и т.д. Примерами слов являются

MAX INFORMATIQVE FORTRAN и т.д.

Требуется строго определить свойство: «x предшествует y в алфавитном порядке», где x и *y* – слова, т.е. дополнить фразу «x предшествует y в алфавитном порядке тогда и только тогда, когда...». Важно заметить, что необходимо дать определение, т.е. характеристическое свойство, а не метод решения вида: «берем первые две буквы слов x и y; если первая буква предшествует первой букве y, то ... иначе, если они одинаковы, берем вторые буквы...» и т.д.

[1] Среди переведенных на русский язык книг, представляющих достаточно полные введения в информатику, следует назвать, кроме уже упомянутой работы [Вирт 72], еще и книгу [Бауэр 76]. О современных зарубежных и отечественных ЭВМ, а также об их математическом обеспечении рассказано в книге [Королев 78]. – *Прим. перев.*

ГЛАВА II.
ВВЕДЕНИЕ В ЯЗЫКИ ПРОГРАММИРОВАНИЯ ФОРТРАН, АЛГОЛ W, ПЛ/1

Его не трогали ни смерть
Его детей, ни предков боль,
Презрев земную круговерть,
Он выбрал Регул и Алголь.
С цветами дев он не ровнял –
Их нрав изменчивый и нежный
Он на кипящий и мятежный
Вулкан науки променял[1]

Раймон Кено
Малая карманная космогония

ВВЕДЕНИЕ В ЯЗЫКИ ПРОГРАММИРОВАНИЯ ФОРТРАН, АЛГОЛ W, ПЛ/1

Главное орудие программиста – его средство выражения: язык программирования. Действительно, исчерпывающее изучение языков приводит к рассмотрению принципиальных проблем программирования.

Целью этой главы является не исчерпывающее исследование языков, а лишь обзор основных элементов используемых языков, и в частности трех из них – ФОРТРАНа, АЛГОЛа W и ПЛ/1. Эти основные элементы будут необходимы при рассмотрении более глубоких свойств тех же языков – управляющие структуры, структуры данных, рекурсивные выражения, – эти исследования будут продолжены в гл. III–VI.

II.1. Основные характеристики

II.1.1. Значения и типы

Цель программы состоит в вычислении *значений.* В свою очередь вычислительная машина оперирует не со значениями, а скорее с *представлениями* значений, которые являются конфигурациями битов, байтов или слов памяти. Так как физические представления зависят от изображаемых объектов, то для того, чтобы оперировать со значениями, необходимо специфицировать их *типы.* Это кажущееся ограничение может оказаться на деле выгодным, потому что оно заставляет программиста точно определить все используемые им объекты и лучше контролировать свою программу.

Ниже перечислены некоторые часто используемые типы вместе с основными понятиями об их представлениях, которые могут быть полезны программисту.

[1] Перевод А. В. Тарновского.

II.1.1.1. Литеры

Чтобы изобразить литеры, принадлежащие заданному «алфавиту» (обычному алфавиту с прописными и строчными буквами, который расширен десятичными цифрами и какими–нибудь специальными символами, как $ @ < и т.д.), используют коды литер (наиболее распространенными кодами являются коды BCD, EBCDIC, ASCII). В зависимости от кода литера изображается 6 битами, 7 битами или 8 битами (байтом).

II.1.1.2. Строки[1]

«Строка», или «цепочка литер» (английское *STRING*) – это конечная последовательность литер, например

АБРА КАДАБРА

Заметим, что объект типа «литера» это частный случай «строки» (последовательность длиной в одну литеру). Используются же они, вообще говоря, по–разному.

Внутреннее представление «строк» будет рассмотрено в II.3.3.5 в связи с ФОРТРАНом.

II.1.1.3. Логические значения

«Логическое» значение представляет собой одно из двух значений *истина* или *ложь*; логические значения появляются всякий раз, когда окружение программы оказывает влияние на ход ее выполнения (*условные операторы,* ср. ниже III.4.3) или на вычисление выражения (*условные выражения,* II.1.2.4).

Логическое значение может быть представлено состоянием бита. Однако трансляторы по соображениям, связанным со способом адресации, ставят часто в соответствие логическому значению состояние литеры или слова.

II.1.1.4. Целые числа

Вычислительные машины разрешают работать с целыми числами, положительными, нулевыми или отрицательными. Множество Z целых чисел в математике бесконечно, но в вычислительной машине целое изображается, вообще говоря, фиксированным числом битов; если число битов равно n (где n чаще всего длина машинного слова), это означает, что в машине представимы только 2^n целых чисел. Так, на ИБМ 360 или 370 n = 32 и представимые целые числа содержатся между -2^{31} и $+2^{31}-1$, т.е. приблизительно ±2 миллиарда; для изображения этих чисел используется соответствующий код.

Заметим, что машинные операции, которые имитируют арифметические действия +, —, × и :, могут порождать *переполнения,* даже если оба операнда находятся в разрешенном интервале (например, в ИБМ при попытке сложить два числа, превосходящих или равных 2^{30}). В зависимости от машин и языков программа может располагать или не располагать средствами обнаружения такой ошибки.

II.1.1.5. Дробные числа («вещественные»)

Термин «вещественное», часто используемый в программировании как противопоставление термину «целое», в сущности, неправилен; он обозначает число, имеющее дробную часть, но в машине можно представить только *конечное* подмножество множества *рациональных* чисел.

---

[1] Авторы предпочитают называть этот тип значений «текстами»; в переводе тем не менее сохранено устоявшееся в русской литературе название. – *Прим. перев.*

Обычные человеческие обозначения используют для дробных чисел одну из двух форм:

±a,b

$+ a, b \times 10^n$ («научная нотация»)

Представление в машине обобщает вторую из этих нотаций. Дробное число относительно некоторой системы счисления рассматривается состоящим из знака (+ или —) и пары

$$[c, n]$$

изображающей число $x = c \times B^n$. Здесь **B, основание** системы счисления, фиксированное целое положительное число; c, **мантисса** числа x, положительное целое число, заключенное между 1/B и 1, или 0 (для изображения дробного числа x = 0) и изображаемое фиксированным числом S битов (таким образом могут быть изображены 2^s различных значений); наконец, n – порядок x, это целое положительное, отрицательное или 0, заключенное между двумя крайними значениями MIN и MAX; (часто имеет место MIN = –MAX).

Машинная арифметика вещественных чисел полностью характеризуется четырьмя целыми константами: B (основание), S (число значащих битов), MIN и MAX (предельно возможные значения порядка).

Так, машина ИБМ серии 360 или 370 использует систему счисления с основанием B = 16 («шестнадцатеричная» система), MIN = — 64 и MAX = 63. Фактически на этих машинах есть два способа представления чисел;

- так называемые вещественные *простой точности* используют 6 значащих шестнадцатеричных цифр (S = 4 x 6 = 24, поскольку нужно 4 бита для изображения цифры по основанию 16);

- вещественные *удвоенной точности* используют 14 значащих шестнадцатеричных цифр (S = 56). Следовательно, на ИБМ 360/370 можно изобразить $15\times2^{28} + 1$ различных «вещественных» чисел простой точности и $15\times2^{60} + 1$ «вещественных» удвоенной точности. Для многих задач точность в 6 шестнадцатеричных цифр (т.е. выше чем 7 десятичных цифр) недостаточна, поэтому рекомендуется систематически применять удвоенную точность.

Всякое вещественное число из интервала $[-B^{MAX}, +B^{MAX}]$ может быть приближено некоторым точно представимым числом, т.е. в форме

$$x = +c \times B^n$$

($MIN \le n \le MAX$; c изображается S битами и $c = 0$ или $1/B<c< 1$).

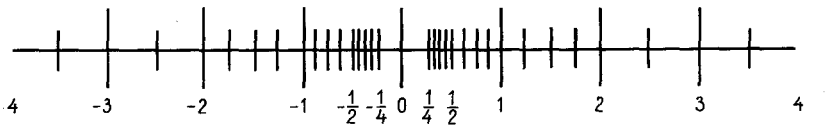


Рис. II.1 Машинная арифметика вещественных (основание 2, MIN = 1, MAX = 2, S = 3).

Рис. II.1 дает представление о множестве точно изображаемых чисел. Здесь «нарисована» «миниатюрная» система, в которой основание B = 2 (двоичная система), MIN = –1, MAX = 2 и S = 3 значащих бита. Как во всякой системе счисления, существует одинаковое количество точко изображаемых чисел в каждом из интервалов

$(B^{MIN}, B^{MIN+1}], (B^{MIN+1}, B^{MIN+2}], \ldots, (1, B], (B,B^2], \ldots, (B^{MAX-1}, B^{MAX}]$

Как и в случае целых чисел, моделируемые арифметические операции могут приводить к переполнению. Для вещественных чисел добавляется, кроме того, риск «недополнения», когда при выполнении умножения (или в исключительных случаях вычитания) результат оказывается меньшим по абсолютному значению, чем наименьшее изображаемое положительное число.

В принципе всякое вещественное число x из отрезка $[-B^{MAX}, B^{MAX}]$ представимо с помощью числа x, самого близкого во «множестве точно представимых чисел». Напротив, моделируемые арифметические .операции вполне могут и не давать

«наилучший» возможный результат. Например, если $\oplus$ есть операция «сложения», выполняемая машиной, может случиться, что

$$\overline{x} \oplus \overline{y} \neq \overline{x + y}$$

С практической точки зрения программист должен знать, что обработка «вещественных» чисел на машине требует введения *приближения.* Весьма наглядный даже для непосвященного читателя случай приведен в II.4.4.2, где «оператор печати», требующий напечатать константу, такую, как 2.718281, вызывает выполнение программы печати близкого, но отличающегося значения. Следует заметить, что арифметические операции, фактически исполняемые машиной, не проверяют свойства, требуемые соответствующими математическими операциями. Так, сложение и умножение не являются ассоциативными: операции

$$(a \oplus b) \oplus c$$

и

$$a \oplus (b \oplus c)$$

не дают, вообще говоря, один и тот же результат (с другой стороны, коммутативность выполняется). Точно так же нет единственного нейтрального элемента для сложения и умножения: афх вполне может иметь то же значение, что и а, но х при этом будет не равным нулю (например, если а = 1, а х – ненулевой элемент по системе счисления, но меньше «машинного нуля»).

II.1.1.6. Комплексные числа

Языки, используемые для научно–технических расчетов, разрешают работу с «комплексными числами», изображаемыми парами дробных чисел; например, [3.17, –2.50] изображает число 3,17 –2,50i

II.1.2. Основные объекты: константы, переменные, массивы, выражения

Каждый программный объект имеет, как мы видели, *тип* и *значение.* Он должен иметь также *имя* в программе.

II.1.2.1. Константы

Константа имеет фиксированное имя, фиксированный *тип* и фиксированное *значение.* В большинстве языков программирования, например, обозначение

134

есть имя константы типа «целое» и фиксированного значения 134 (сто тридцать четыре). Точно так же константа типа «строка», значением которого является строка, образованная из восьми литер С, В, Е, Т, Л, А, Н, А в указанном порядке, будет обозначаться *'СВЕТЛАНА'* в ПЛ/1, *"СВЕТЛАНА" в* АЛГОЛе и *8НСВЕТЛАНА* в стандартном ФОРТРАНе.

Для таких констант *тип* и *значение* выводятся непосредственно из *имени.* Некоторые языки позволяют также программисту выбрать другое имя для обозначения константы. Например, программист может пожелать связать с константой «дробного» типа и значением 3.141592 имя *ПИ* В таких случаях говорят о **символических константах** (т.е. таких, которые могут обозначаться некоторым символом вместо принятого для константы обозначения объявлением ее значения).

Использование символических констант – это хороший прием программирования: он исключает ситуации, при которых значения, являющиеся по существу параметрами выполняемой программы, представляют в явной форме многократно в разных местах программы. Такие ситуации осложняют модификации и расширения программ. Значение символической константы появляется только в одном месте, в *объявлении символической константы,* которое позволяет связать значение константы с выбранным именем; если появляется необходимость перейти к другому значению, модифицируется только это объявление. В некоторых языках программирования строго применяется этот принцип, запрещая использование констант,

имеющих отличную от символической форму.

II.1.2.2. Переменные

Переменная имеет фиксированное *имя,* фиксированный *тип* и переменное *значение* в соответствии со спецификациями программиста.

Имя переменной называется **идентификатором.** Вот примеры идентификаторов, допустимых в трех наших языках:

X I ABRA MEYER BAUDOI AB6 G7

Тип переменной связывается с ее именем с помощью **объявления типа,** которое может быть неявным.

Переменная может получать значение в ходе выполнения программы среди прочих способов путем *присваивания* или *чтения* (см. далее, II.1.3).

В заключение определим *переменную* в информатике как триплет [идентификатор, тип, значение], где только третий элемент является переменным[1]. Заметим, что это понятие немногим отличается от понятия переменной в математике (математические «переменные» это, вообще говоря, априорно неизвестные, нефиксированные величины, и их правильнее было бы называть *неизвестными*).

II.1.2.3. Массивы

Массив имеет фиксированное *имя,* фиксированный *тип* и *много значений,* связанных с этим массивом *переменных.* Интерес к понятию массива объясняется тем, что оно позволяет изобразить совокупности объединяемых значений, например *матрицы* из математики. Каждое из значений указывается именем, образованным из имени массива и одного или несколько *индексов;* с таким именем можно работать как с именем переменной. Индексы имеют значения, принадлежащие конечному множеству, как правило, подмножеству целых чисел.

Имя массива является идентификатором.

Объявление массива уточняет его тип (который может задаваться неявно), число его *измерений* и *границы* каждого измерения.

Рассмотрим сначала *одномерные* массивы. Две границы и M – целые, такие, что $m \le M$. Этот массив эквивалентен, таким образом, набору $M - m + 1$ переменных, для которых разрешенными именами являются ТАБ[i] (или ТАБ(i) в зависимости от языка), где ТАБ – имя массива, а i задает целое значение, такое, что $m \le i \le M$ Говорят, что ТАБ[т], ..., ТАБ[M] – элементы массива ТАБ.

В случае массива n измерений $n > 1$ каждое измерение имеет пару соответствующих границ $[m_i, M_i]$ и элементы массива обозначаются ТАБ $[a_1\ a_2, .., a_n]$, где каждое a_i означает целое, заключенное между m_i и M_i.

Отметим, что любое объявление массива определяет две категории имен: имя массива в целом, которое обрабатывается как имя переменной с ограничениями, меняющимися от языка к языку, и формируемые имена для обозначения каждого элемента массива («индексируемые переменные»). В некоторых условиях можно также с помощью имени массива сформировать имена *подмассивов* (IV.4.5).

II.1.2.4. Выражения

Выражение позволяет обозначить вычисление каждого значения по другим значениям и операциям. Входные значения выражения могут быть константами, значениями переменных или элементов массивов, или значениями выражений. Так, в наших трех языках примерами выражений типа ЦЕЛОЕ могут быть:

[1] В некоторых языках *тип* является переменным.

3+27

X + 3 (если *X* – имя целой переменной),

T(2) + 7 (если *T* – имя массива целых с границами 1 и 3),

*3*20* (что значит *60;*, * изображает умножение),

*A + B*C* (если *A B C* – имена целых переменных).

В таких выражениях, как в последнем примере, порядок операций (как и в математике) определяется их *приоритетом:* говорят, что умножение имеет более высокий приоритет, чем сложение, чтобы указать, что при отсутствии скобок A + B * C обозначает сумму значения A и произведения B * C, а не произведения суммы A + B на C. Мы увидим правила приоритетов в различных языках, которые в целом близки к общепринятым соглашениям. В сомнительных случаях или для того, чтобы нарушить правила приоритета, можно всегда использовать скобки:

A + (B*C) (A + B)*C

Кроме выражений числовых типов (целые, вещественные), языки программирования позволяют оперировать с выражениями типа «строка», получаемыми, в частности, с помощью операторов, работающих с текстами: выделение подстроки, конкатенация (соединение) двух строк и т.д.

Точно так же существуют выражения логического типа, т.е. обозначающие истину или ложь. Они, в частности, составлены из операторов отношений, позволяющих проверять равенство или неравенство двух объектов или отношения порядка, определенные на числах или на строках (алфавитный порядок). Так, если *I* и *J* – целые переменные, то с помощью отношения *I < J* в АЛГОЛЕ W и ПЛ/1 или *I.LT.J* в ФОРТРАНе обозначают выражение логического типа (так называемого типа BIT в ПЛ/1), означающего истину, если *I* меньше *J,* ложь в противном случае.

Заметим по этому поводу, что в силу приближений, возникающих благодаря представлению дробных чисел в машине (ср. выше II.1.1.5), логическое выражение, сравнивающее два объекта a и b типа ВЕЩЕСТВЕННЫЙ с помощью операторов равенства или неравенства, не имело бы большого смысла; например, такое логическое выражение, как *2.71=(5.42/2)* или еще *2.71.EQ.(5.42/2.)* в обозначениях ФОРТРАНа, вполне может иметь ложное значение. Важнее сравнивать абсолютное значение |a — b| разности двух чисел с малой положительной константой ε. Исключением из этого правила является сравнение с числом нуль, которое дает правильный результат.

II.1.3. Программы, операторы

Цель программы состоит, вообще говоря, в вычислении одного или нескольких значений. Хорошо было бы уметь задавать программу в виде формулы, в которую входили бы данные и операции обработки этих данных, т.е. в виде *выражения* (термин, который мы только что определили). Тогда программа, вычисляющая сумму квадратов двух чисел a и b, могла бы записываться просто

a^2+b^2

или

сумма (квадрат (a), квадрат (b))

а вызов программы автоматического перевода можно было бы представить в виде

русско–французский–перевод ("ВОЛК И СЕМЕРО КОЗЛЯТ")

В наших трех языках программирования нельзя специфицировать программы таким образом, и мы обязаны сами задавать этапы «вычисления» списком предписаний, называемых **операторами.** Правильные совокупности операторов образуют **программы,** разрешенные для рассматриваемого языка. Различают два типа операторов:

- операторы *обработки информации* в собственном смысле слова,
- операторы, которые *управляют* работой программы во времени.

Операторы второго типа, называемые «управляющими структурами», будут рассматриваться в следующей главе. Сейчас мы ограничимся тремя видами операторов собственно обработки информации:

а) оператор присваивания имеет дело с переменной или элементом массива, например x, и выражением, например e. Он записывается в зависимости от языка либо x←e, либо x:=e, либо x = e (и даже e→x).

Записывая этот оператор, программист полагает, что по окончании его выполнения x будет принимать в качестве нового значения величину (значение) выражения e. Обычно x и e имеют одинаковый тип (в ПЛ/1 они могут быть массивами).

Характеристическое свойство присваивания x←e, смысл которого определится в разд. III.4, состоит в том, что всякое отношение P переменных программы будет истинным после выполнения этого присваивания, если и только если P [e→x] было истинно до выполнения; последнее обозначение выражает замену всех вхождений переменной x на выражение e. Можно убедиться в том, что это свойство верно, независимо от того, присутствует ли x в выражении e (кроме частного случая «побочного эффекта»; см. гл. IV).

б) Оператор **чтения** определяет, что новое значение переменной должно быть *прочитано* с указываемого периферийного устройства и присвоено указываемой переменной. Чтение имеет преимущество перед присваиванием константы, так как делает программу инвариантной по отношению к данным; одна и та же программа может выполняться с разными данными.

в) Наконец, оператор **записи** предписывает передачу значения выражения, переменной или константы на периферийное устройство.

II.2. Алгоритмическая нотация

Мы вкратце опишем сейчас алгоритмическую нотацию высокого уровня. Договоримся: мы далеки от мысли прибавить еще один новый язык к нескольким сотням, имеющимся уже на «рынке»! Мы просто хотим ввести некоторые средства выражения, которые позволят нам изображать алгоритмы, не затрагивая особенностей ФОРТРАНа, АЛГОЛа W или ПЛ/1 (или какого–нибудь другого языка). Единственное превосходство нашего «псевдоязыка» над существующими языками состоит в том, что его описание помещается на нескольких страницах.

Наш язык (знаки которого отмечаются цветными литерами алфавита, включающего латинские и русские буквы[1]) имеет свободный формат, т.е. мы допускаем свободную вставку и удаление пробелов и переходов со строчки на строчку для лучшей читаемости программ. Его синтаксис характеризуется тем, что программы этого языка предназначаются для чтения их людьми, а не трансляторами. Основные обозначения описаны ниже; другие будут введены в следующих главах. Сводный справочник. свойств «языка» можно найти в приложении А.

В любое место программы могут вставляться *комментарии,* заключаемые в фигурные скобки { и }, например

{это комментарий}

Комментарии не предписывают выполнение какого–либо действия; они будут служить средством, делающим программы более ясными и убедительными за счет включения пояснений, в частности *утверждений,* отмечающих свойства, которые необходимо проверять на некоторых этапах выполнения программы.

---

[1] Оригинал ограничивается, естественно, только латинскими буквами. – *Прим. перев.*

Наш язык работает с целыми, вещественными, логическими типами, с литерами и строками; константа типа СТРОКА заключается в кавычки, например

"ЭТО СТРОКА"

Константа типа ЛИТЕРА это «строка», состоящая из одной литеры, например "А". Мы будем иногда пользоваться типом ЦЕЛОЕ НАТУРАЛЬНОЕ (целое положительное и нуль).

Будем использовать прописные и строчные литеры. Прописные предназначаются для констант типа «строка» и имен типов: ЦЕЛОЕ, СТРОКА и т.д. Наш псевдоязык имеет *ключевые слова,* которые записываются полужирными буквами, например **истина**, **ложь**, **переменная**, **повторять**, **если** и т.д.

Благодаря ключевому слову консгинта можно объявлять символические константы, например

константа золотое сечение = 1.618034

Выражения записываются обычными математическими обозначениями

$$a + \frac{b}{c} * (u + v - 27)^3$$

с той разницей, что в дробных числах точка заменяет запятую, как в 3.75 или -27.50×10^6. Деление изображается горизонтальной или наклонной чертой.

Будем использовать *условные выражения,* обозначаемые

если c **то** e_1 **иначе** e_2

где c – логическое выражение, e_1, и e_1 – выражения одного типа (любого). Например, если x и y оба имеют тип ЦЕЛОЕ, значение выражения

если x > y **то** x **иначе** y

есть максимум значений x и y[1]

Два специальных значения, обозначенные $+\infty$ и $-\infty$, изображают соответственно самое большое и самое малое числа из изображаемых в машине.

Всякая *переменная* должна быть объявлена. *Объявление* имеет вид

переменная a: ЛИТЕРА {или ЦЕЛОЕ, или СТРОКА, или ЛОГИЧЕСКОЕ,
или ВЕЩЕСТВЕННОЕ }

Разрешается свободное использование формы множественного числа и всех трех родов – мужского, женского и среднего – для указания типов[2]:

переменные u, v, w : ЦЕЛЫЕ

Кроме того, разрешаются сокращения при указании типа – ЛОГ, ЦЕЛ, ВЕЩ.

Последовательные объявления отделяются точкой с запятой:

переменные a, b : ЛИТЕРЫ;

переменные x, y : ВЕЩЕСТВЕННЫЕ;

переменные u, v : ЛОГИЧЕСКИЕ

Можно сгруппировать объявления при условии, что объединяются четко различаемые (с помощью смещения) позиции объявлений; в этом случае для отделения разных групп переменных пользуются запятой:

Переменные a, b : ЛИТЕРЫ,
x, y : ВЕЩЕСТВЕННЫЕ,
u, v : ЛОГИЧЕСКИЕ

---

[1] Не путать условные выражения с условными операторами и переключателями(альтернатива, многозначное ветвление), которые вводятся в следующей главе(III.3.2).

[2] В оригинале речь идет только о допущении формы множественного числа. – *Прим. перев.*

Массив объявляется, например, так:

массив таб [–2 : 7]: ЦЕЛЫЙ

или

массив p[0 : 5,1 : 10,1 : 4]: СТРОКА{или любой другой тип}

Здесь – 2 и 7 – границы массива таб; [0, 5], [1, 10] и [1, 4] – границы изменений массива p.

Элемент из таб будет обозначаться таб [i], элемент из p – p[i, j, k].

Присваивание имеет вид

переменная ← выражение

где переменная и выражение имеют одинаковый тип; переменная может быть элементом массива. *Чтение* имеет вид

читать (ф) пер1, пер2,.. .,перn

где пер1, пер2,..., перn – это переменные любых типов, а (ф) – указание на то, что можно опустить, если в этом нет необходимости для понимания программы, обозначение периферийного устройства (читающего перфоратора, файла на дисках,...). Чтение возможно, только если логическое выражение конец файла (ф) имеет значение **ложь**.

Операция *записи* имеет вид

писать(ф) выр1, выр2, ..., вырn

где выр1, выр2, ..., вырn – выражения любых типов, а (ф) может быть опущено.

Вот и все об «основах» нашего псевдоязыка. Другие его обозначения мы увидим в следующих главах.

* *
*

В трех следующих разделах мы приступим к описанию трех языков программирования, важных и интересных, а именно ФОРТРАНа, АЛГОЛа W и ПЛ/1. Это описание будет продолжено в следующих Главах.

В описании этих языков элементы, написанные *ПРОПИСНЫМ ЦВЕТНЫМ КУРСИВОМ,* представляют собой разрешенные объекты, принадлежащие языку. Примеры:

(ФОРТРАН) *IDENT DO G27 GOTO DOUBLE PRECISION CONTINUE*

(АЛГОЛ W) *IDENTIFICATEUR FOR BEGIN END LONG REAL DO*

(ПЛ/1) *IDENTIF PROCEDURE DO BINARY FLOAT*

Элементы, записанные *строчным цветным курсивом,* представляют объекты, разрешенные в языке. Так, когда мы пишем на АЛГОЛе

IF логическое выражение THEN оператор

это означает «все конструкции, получающиеся включением некоторого 'логического выражения' перед словом АЛГОЛа W *THEN* и некоторого 'оператора' после этого слова»; определения терминов 'логическое выражение' и 'оператор' считаются при этом известными. Наконец, тексты, отделенные горизонтальными линиями и сопровождаемые указанием языка ФОРТРАН, АЛГОЛ W или ПЛ/1 в левом верхнем углу, представляют собой программные модули, разрешенные в рассматриваемом языке.

Пример[1]:

ПЛ/1

```
/*AMICAL – ПРИВЕТСТВИЕ */
AMICAL: PROCEDURE OPTION (MAIN);
        PUT LIST ('ЗДРАВСТВУЙТЕ');
        END AMICAL;
```

Описания языков бывают порой скучными; мы попытались взять из ФОРТРАНа, АЛГОЛа W и ПЛ/1 только по–настоящему полезные и важные элементы. Мы советуем читателю бегло познакомиться при первом чтении с представленными здесь тремя языками и концепциями программирования, которые они отражают. Насколько пагубно путать обучение программированию с изучением подробностей языка, настолько же опасно не знать главных механизмов распространенных языков программирования для описания алгоритмов. Даже недостатки языков программирования говорят зачастую о наличии концептуально важных проблем.

II.3. Введение в ФОРТРАН

II.3.1. История

Название языка «ФОРТРАН» есть сокращение "FORmula TRANslation" (перевод формул): ФОРТРАН был одним из первых языков, позволивших кодировать арифметические вычисления в одном–единственном операторе с помощью формул, похожих на принятые в математике. Кодирование формул было, впрочем, первоначальной целью проекта ФОРТРАНа; только спустя некоторое время его разработчики заметили, что совсем нетрудно указать в той же системе порядок вычислений, зафиксировав тем самым рождение первого языка программирования широкого распространения.

ФОРТРАН родился в 1954 г. в ИБМ, и его первая версия ФОРТРАН I могла уже использоваться с 1957 г. на машине ИБМ 704. Серия модификаций и дополнений породили затем ФОРТРАН II в 1958 г. и ФОРТРАН IV в 1962 г. Определение языка было стандартизовано в 1966 г. Американской ассоциацией стандартов (АСА), которая теперь называется АНСИ [ANSI 66]. С тех пор трансляторы ФОРТРАНа создаются для большинства вычислительных машин, кроме некоторых самых малых.

Можно спросить, почему ФОРТРАН со своим более чем двадцатилетним стажем выжил и даже остается вместе с КОБОЛом самым употребляемым языком. Одна из причин, несомненно, заключена в явной простоте языка; к сожалению, за этой простотой часто скрываются практические трудности, многочисленные конкретные случаи, всевозможные исключения, которые надо помнить; об этом часто будет упоминаться в этой книге, и это обязывает программиста иметь про запас несколько определенных приемов, чтобы удобно использовать ФОРТРАН. Действительно,

[1] В выбранных авторами языках алфавиты, в которых описываются идентификаторы, включают только латинские буквы, поэтому ни в ФОРТРАНе, ни в АЛГОЛе W, ни в ПЛ/1 такой идентификатор, как, например, *ИДЕНТИФИКАТОР*, строго говоря, недопустим, так как он составлен из букв не принятой в этих языках кириллицы. Поэтому в примерах программ перевод идентификаторов включен в комментарии, записанные в соответствии с правилами каждого языка (в комментариях разрешено использование, вообще говоря, любых литер).
Тексты, заключенные в кавычки и являющиеся аргументами операторов печати, включают любые литеры, допустимые во внутреннем алфавите машины и на внешних печатающих устройствах (т.е. в условиях отечественных машин и буквы русского алфавита, как правило). Поэтому предусмотренные для печати тексты всюду, где это помогает пониманию программ, набраны по–русски с полным уважением правил рассматриваемых языков программирования. – *Прим. перев.*

живучесть этого языка, по–видимому, связана с экономическими проблемами, которые возникают с переработкой существующих важных «библиотек программ», переписыванием «оптимизирующих» трансляторов, которым были посвящены значительные усилия, и т.д.

Следует добавить, что ФОРТРАН достиг высшей степени стандартизации по сравнению с другими языками широкого распространения; к сожалению, официальный стандарт языка имеет существенные пробелы (обработка литер, числовые свойства), и почти все разработчики предлагают версии ФОРТРАНа, «улучшенные» многочисленными расширениями, самый очевидный эффект которых состоит в увеличении риска при любом переносе программы с одной машины на другую. Настоятельно рекомендуется придерживаться стандарта языка, чтобы не потерять главное преимущество использования ФОРТРАНа. В этой книге мы ограничимся в основном заключенными в апострофы цепочками литер, без которых действительно было бы слишком трудно работать и которые допускаются почти всеми трансляторами.

В США подготавливается новый стандарт ФОРТРАНа [ACM 76a]; он должен улучшить некоторые из спорных пунктов языка, но тем самым увеличит число основных понятий и сложность. В момент написания этой книги (1976–1977 гг.) непохоже, что отладка первых соответствующих этому стандарту трансляторов должна завершиться в ближайшие годы.

II.3.2. Значения и типы

В ФОРТРАНе существуют следующие типы:

- *INTEGER* (целое),
- *REAL* (вещественное),
- *DOUBLE PRECISION* («вещественное» с повышенной точностью),
- *LOGICAL* (логическое),
- *COMPLEX* (комплексное),
- строка (тип, определенный частично: есть константы этого типа, но нет переменных).

II.3.3. Основные объекты языка

II.3.3.1. Константы

Целая константа записывается в обычной десятичной форме:

0 –3 45678 +5 и т.д.

Знак + не обязателен для положительных чисел.

- *Вещественная константа* (простой точности) записывается с точкой (которая является англо–саксонским эквивалентом нашей десятичной запятой), отделяющей целую часть от дробной; эта точка может присутствовать в начале числа, в конце его или посредине. Можно сопровождать число показателем, предписывающим умножение на степень числа 10; показатель состоит из буквы *E,* знака (+ может быть опущен) и числа из одной или двух цифр. Например,

3.14159 .3145159E+1 .0314159E02 314159.E–5

это четыре способа записать приближенное значение числа π. Последнее число, например, читается «314159, умноженное на десять в степени минус пять».

- *Вещественная константа удвоенной точности* записывается с точкой и обязательным показателем, который использует букву D; например,

3.1415926535898D0 3141592635898D+1

это две приближенные записи π, лучшие, чем предыдущие.

- *Комплексная константа* записывается в виде двух вещественных констант простой точности, которые представляют вещественную и мнимую части комплексного числа соответственно; они помещаются в круглые скобки и разделяются запятой. Например,

(3.1.–4E–3) представляет число 3,1–0,004i;

(–.1E3, 3E2) представляет число –100 + 300i.

- *Логическая константа* записывается в одной из двух форм *.TRUE.* или *.FALSE.* что означает «истина» и «ложь» соответственно. Заметим, что точки составляют неотъемлемую часть этих двух символов.

- *Константа типа «строка» или «литера»* это последовательность из одной или нескольких литер, заключенная между двумя апострофами. Примеры:

'ЕВГЕНИЙ ОНЕГИН'

'A'

Если одна из литер строки есть литера «апостроф», то в изображении строки она фигурирует как два смежных апострофа:

'L''ELISIR D''AMORE' (строка, содержащая литеры L'ELISIR D'AMORE)

' ' ' ' (строка, содержащая только одну литеру – апостроф).

Использование констант типа «строка» подчинено в ФОРТРАНе серьезным ограничениям, которые мы рассмотрим ниже (II.3.3.5).

Способ описания, использующий апострофы, в действительности не допускается стандартом АНСИ. Официально текст, образованный из литер c_1, c_2, ..., c_n, изображается

$nHc_1\ c_2, \ldots\ c_n$

Например, *16НЕВГЕНИЙ ОНЕГИН*

1HA

16H L'ELISIR D'AMORE

1H'

Мы ограничимся в этой книге (это наше единственное нарушение стандарта ФОРТРАНа) использованием формы с апострофами, допускаемой, впрочем, большинством трансляторов: не имея под рукой вычислительной машины, мы отказываемся отсчитывать каждый раз литеры.

II.3.3.2. Переменные. Символические константы

В ФОРТРАНе идентификаторы образуются последовательностями, включающими от одной до шести литер, первая из которых обязательно должна быть буквой, а остальные – буквами или цифрами. Так, следующие имена в ФОРТРАНе правильны:

A Z TOTAL A12B34 NPLUS1

Ограничение шестью литерами затрудняет иной раз подбор ясных и содержательных имен.

Объявления переменных, которые позволяют связывать идентификатор (имя) с типом, иллюстрируются следующими примерами и комментариями:

INTEGER A	целое
REAL B, B1, B2	вещественное простой точности
DOUBLE PRECISION C	вещественное удвоенной точности
COMPLEX D, D1	комплексное
LOGICAL E, F	логическое

Можно опустить объявление целого, если соответствующий идентификатор

начинается с одной из букв *I ,J, K, L, M* или *N,* а также можно опустить объявление вещественного простой точности, если идентификатор начинается с одной из 20 других букв латинского алфавита.

Тот факт, что в ФОРТРАНе разрешается делать объявления «по умолчанию», в силу чего всякий необъявленный идентификатор неявно рассматривается в качестве имени целой или вещественной переменной (в зависимости от первой буквы), лишает трансляторы возможности выявлять некоторые простые ошибки: если в идентификаторе допущена орфографическая ошибка (например, *MONOM* вместо *MONOME*), транслятор будет считать, что речь идет о новой переменной, начальное .значение которой, очевидно, не определено. Таким образом, тривиальные ошибки могут заметно усложнить отладку программ на ФОРТРАНе. Мы всегда будем явно объявлять все переменные и рекомендуем придерживаться этого правила.

Особого рода предписание *DATA* позволяет присвоить начальные значения одной или нескольким переменным. Можно, например, объявить

```
DOUBLE PRECISION PI, E, AVOGAD
INTEGER POLE, MORE
```

и включить следующие предписания:

```
DATA PI/3.1415926536D0/, E/2.7182818285D0/
DATA AVOGAD /6.0225D23/, MORE /1515/, POLE /7/
```

Хотя *DATA* может служить для инициализации переменных, следует, вообще говоря, как подсказывают эти примеры, оставлять это предписание для объявления имен, которые желательно связать с «символическими константами», как *POLE* или *MORE* в приведенном выше примере. Предписание *DATA,* которое не надо путать с оператором присваивания, обращается к транслятору и выполняется только один раз, даже если программа выполняется многократно. Инициализация переменных (которая должна повторяться при каждом выполнении) это функция операторов присваивания[1].

II.3.3.3. Массивы

В ФОРТРАНе можно определять массивы одного, двух и трех измерений (некоторые трансляторы допускают и большее число измерений). Это делается путем включения верхних границ по каждому измерению в объявление типа массива; нижняя граница всегда неявно считается равной 1. Таким образом, можно объявить

```
INTEGER A (10,20)
LOGICAL TERMIN, VID(50)
```

В последнем объявлении только *VID* является массивом. Можно также отделить объявление границ от объявления типа с помощью особого предписания *DIMENSION.* Например,

```
INTEGER A
LOGICAL TERMIN, VID
DIMENSION A(10,20), V1D(50)
```

В обоих случаях *A* – это массив из 10 × 20 = 200 элементов. Возможно, читателям покажется интересным, что стандарт ФОРТРАНа задает порядок размещения массива в памяти; элементы упорядочены «по столбцам», т.е. для нашего массива *A* следующим образом:

```
A(1,1),A(2,1),...,A(10,1),  A(1,2),...,A(10,2),.....,A(1,20),...,A(10,20)

   1 столбец            2 столбец              20 столбец
```

[1] Случай, когда *DATA* полезна для инициализации переменных, имеет место для «остаточных» переменных в подпрограммах (гл. IV, разд. IV.6).

Этот пример обобщается на случай большего числа измерений: первый индекс меняется быстрее всех других индексов, затем второй и т.д.

II.3.3.4. Выражения

Для формирования выражений из констант и переменных используются следующие знаки операций:

а) *арифметические* операции: +, –, *, /, **, которые дают числовой результат при также числовых операндах (или «арифметических»), т.е. *INTEGER, REAL, DOUBLE PRECISION* или *COMPLEX*.

- '*' означает умножение.
- '/' означает деление: если оба операнда деления (константы или переменные) – *целые,* то результат есть целое частное деления: так, *17/5* дает *3*, тогда как *17./5.* дает *3.4*; или еще *1/2* равно нулю, но *1./2.* дает *0.5*.
- '**' означает возведение в степень.
- '–' означает одновременно «минус» вычитания и «минус», который служит для того, чтобы взять противоположное значение величины (их иногда называют соответственно «минус бинарный» и «минус унарный», потому что они применяются по–разному: первый – к двум операндам, а второй – только к одному операнду).

Заметим, что все эти знаки операций (кроме **) могут связывать операнды типа *COMPLEX;* они представляют сложение, вычитание, умножение и деление комплексных чисел в том смысле, в котором эти операции определены в математике.

Выражение должно иметь операнды одинакового типа и обладать в силу этого тем же типом.

Есть два исключения из этого правила:

– выражение

*r**e*

где *r* – вещественное («простое» или «удвоенное»), а *e* – целое, допустимо, и оно имеет тот же тип, что и *r*;

– можно комбинировать операнды типа *REAL, DOUBLE PRECISION* и *COMPLEX,* В таком случае результирующее выражение имеет тип *DOUBLE PRECISION* или *COMPLEX,* если по крайней мере один из операндов имеет тип *COMPLEX*.

Некоторые трансляторы идут дальше и разрешают различные «смешанные» выражения. Этими возможностями не следует пользоваться: они меняются от транслятора к транслятору и делают программу трудно переносимой с одной машины на другую; кроме того, вычисление смешанного выражения вызывает преобразование типов одного представления в другое. Эффект такого преобразования не всегда очевиден. Если необходимо комбинировать значения разных типов, тогда надо явно упоминать эти преобразования с помощью *функций преобразования типов* ФОРТРАНа (см. ниже II.3.4.3.а).

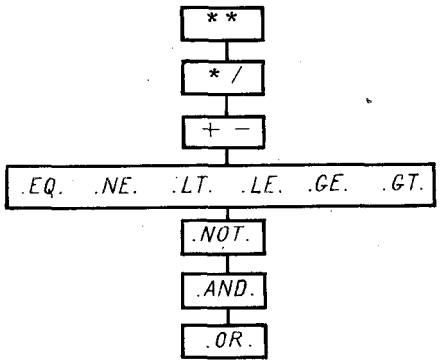
б) операторы *отношений,* которые дают *логические* результаты исходя из *арифметических* операндов. Их шесть:

символ ФОРТРАНа	словесное выражение	математический символ
.GT.	больше	>
.GE.	больше или равно	≥
.EQ.	равно	=
.NE.	не равно	≠
.LE.	меньше или равно	≤
.LT.	меньше	<

Точки являются неотъемлемой частью этих символов, что не облегчает чтение некоторых выражений. Так, *3.0.LE..3E1* разделяется на один операнд, *3.0* , знак операции *.LE.* и другой операнд, *.3E1*. Выражение имеет, следовательно, значение *.TRUE.* *С* методологической точки зрения напомним, что операция *.EQ.* в принципе не рекомендуется для операндов типа *REAL* (II.1.1.5).

в) *логические* операции: *.AND.*, *.OR.*, *.NOT.*, которые дают результаты логического типа, исходя из операндов, тоже логических. Эти операции имеют такие же определения, как и в математической логике*: A .AND. B* означает *.TRUE.*, *A .OR. B* тогда и только тогда, когда оба операнда *A* и *B* имеют значение *.TRUE.* , *A .OR. B* равно *.FALSE.* тогда и только тогда, когда оба операнда *A* и *B* имеют значение *.FALSE.* и *.NOT. A* имеет значение, противоположное значению *A* .

г) *Приоритет операций.* В ФОРТРАНе имеет место следующий порядок убывания приоритетов операций:



Когда выражение содержит несколько последовательных операций одного и того же приоритета, как * и / или + и –; или несколько последовательных одинаковых операций, объединение операндов делается слева направо: *7 – 5 + 2* дает 4, а не 0. Поэтому нужно быть особенно осторожным в выражениях вида *A/B*C,* которое эквивалентно *(A/B)*C*, а не *A/(B*C);* напротив, *A/B/C* эквивалентно *(A/B)/C,* а значит, *A/(B*C)* в приближении с точностью до машинного представления, если речь идет о вещественных числах (для которых умножение не строго ассоциативно).

> Не существует проблемы приоритета среди операторов отношения; в самом деле, две смежные операции в выражении не могут быть операторами отношений.

II.3.3.5. Обработка строк в ФОРТРАНе

Абстрактное обсуждение обработки объектов типа «литера» или «строка» в ФОРТРАНе без связи с конкретной машиной не может быть результативным. Тем не менее многие задачи требуют такой обработки, и это оправдывает наше эскизное изложение некоторого общего подхода.

Трудности возникают из–за того, что в ФОРТРАНе нет типов ЛИТЕРА или СТРОКА, определяемых подобно другим типам (*INTEGER, LOGICAL* и т.д.). Переменная или массив, предназначенные принимать значения одного из этих типов, должны быть, следовательно, «закамуфлированы» и объявлены принадлежащими к некоторому другому типу.

> Эта ситуация осложняется полным отсутствием стандартизации в кодировании представлений литер, а также размеров машинных слов. Так, на PDP/10 слово (36 битов) может содержать 5 литер (каждая по 7 литер, код ASCII); на машинах CDC 6600/7600 слово (60 битов) содержит 10 литер (6 битов, код BCD); на ИБМ 360 или 370 слово (4 байтов = 32 бит) содержит 4 литеры (1 байт, код EBCDIC); на большинстве мини–машин слово (16 битов) содержит 2 литеры.

В этих условиях единственное надежное предположение, которое можно сделать, состоит в том, что можно «разместить» литеру на пространстве, отведенном под целое. Действительно, самые богатые коды литер используют 8 битов, и любая машина предлагает по крайней мере $2^8 = 256$ различных целых. Всякое предположение,

идущее дальше этого, рискованно.

> Можно было бы предусмотреть использование другого типа, отличного от целого, для представления литер, но эта возможность исключается полностью, потому что все другие типы ФОРТРАНа осложнены невидимыми программисту преобразованиями; так, вещественные могут оказаться «нормализованными» вопреки желаниям программиста. Тип «целый» в принципе гарантирует от такого рода сюрпризов.

Метод, состоящий в представлении литер с помощью целых, может привести к значительным потерям места в памяти (90% для CDC, где целое занимает слово). Возможны два случая в зависимости от объема обрабатываемого «текста»:

- если литер не слишком много, пытаются облегчить возможный переход с одной машины на другую, сохраняя «транспортабельность» фортрановской программы: в этом случае надо ограничиться одной литерой на целое. Это отвечает общему правилу: адаптация фортрановской программы, использующей конкретные особенности другой машины, к новой машине – весьма нелегкая задача, и не следует приносить в жертву ту разумную дозу «эффективности» в пользу стандартов языка. В VIII.4.6 мы вернемся к проблеме переносимости;

- если, напротив, объем обрабатываемого текста становится слишком большим, приходится частично жертвовать «транспортабельностью». Самый простой метод состоит в том, что все обрабатываемые литеры размещаются в специально выделенной области памяти настолько плотно, насколько это возможно. Тогда подготавливают некоторое число подпрограмм (см. гл. IV), позволяющих всякой фортрановской программе запрашивать создание такой области (в форме массива) и выполнять текущие операции над текстом (формирование, конкатенация двух строк сравнение по алфавиту, выделение подстроки, печать и т.п.).

> Конкретно, строка становится доступной с помощью двух целых, из которых одно представляет адрес начала строки в специальной области, а второе – число подряд расположенных литер. Очень важно сделать так, чтобы подпрограммы, написанные для таких обстоятельств, были единственным средством доступа к этим строкам и чтобы использующие их программы не могли непосредственно воспользоваться свойствами внутреннего представления. Очень важно также разделять эти подпрограммы на такие, которые могут быть написаны на стандартном ФОРТРАНе, и то небольшое число подпрограмм, которые должны использовать особенности машины. Эти последние могут быть написаны на ассемблере либо на ФОРТРАНе, при этом они должны быть снабжены соответствующими комментариями и могут использовать нестандартные свойства. Так, на ИБМ 360 или 370 можно с некоторыми предосторожностями использовать для доступа к байтам тип *LOGICAL*1*
>
> В этих условиях работа, связанная с изменением машины или транслятора, минимальна: переписать несколько подпрограмм, *внешние спецификации* которых остаются неизменными; никакая программа, использующая систему обработки строк, не нуждается в модификации.
>
> Такой набор программ – пример «пакета прикладных программ», назначение которых снабдить программистов возможностями, по тем или иным причинам не предоставленными в их распоряжение системой или языком. Реализация пакетов прикладных программ требует соблюдения стандартов и методов, вообще говоря, более строгих, чем при обычном программировании.

При любом из способов кодирования литер целыми предстоит использовать переменные, представляющие строки или литеры. Их объявляют целыми:

INTEGER CARAC, TEX

После этого можно работать с этими переменными, «как если бы» они имели тип «литера» или «строка». Присваивания таким переменным требуют особой предосторожности (ср. II.3.4.3.а).

II.3.4. Программы, операторы

II.3.4.1. «Работа» ФОРТРАНа и «программные модули»

Программа на ФОРТРАНе состоит из одного или в общем случае из нескольких

«программных модулей», соответствующих логическому разделению выполняемой работы. Один из этих модулей, присутствующий обязательно, это *главная программа;* другие, если они существуют, это *подпрограммы* типа *FUNCTION* или *SUBROUTINE,* которые мы увидим в гл. IV; сейчас нам достаточно знать, что каждый программный модуль заканчивается специальной директивой:

END

которая отделяет его от следующего модуля.

II.3.4.2. Физическая форма программы

Независимо от природы физического носителя программы, на котором она воспринимается машиной, программа на ФОРТРАНе рассматривается сформированной из последовательности 80–литерных строк. Количество литер в строке выбрано по длине перфокарты. Некоторые из этих строк являются *комментариями;* другие содержат программные *директивы,* если сгруппировать под этим термином объявления, операторы и особые предписания *DIMENSION, DATA* или *END*. Ход выполнения программы определяют только директивы.

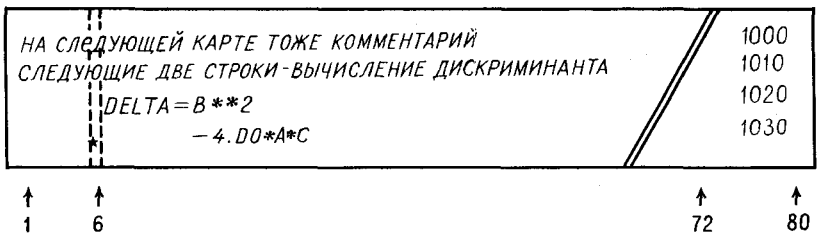
Каждая строка содержит не более одной директивы; с другой стороны, одна директива может быть распределена на несколько последовательно расположенных строк.

Каждая строка директивы содержит четыре части (показанные на рис. ниже):

- **позиции с первой по пятую включительно**: содержат метку директивы, если она есть. Понятие метки будет введено в следующей главе;
- **позиции с 7 по 72 включительно**: содержат собственно директиву. Можно свободно вставлять пробелы между идентификаторами и другими символами языка; это обстоятельство можно использовать, чтобы сделать программу удобной для чтения. В частности, не обязательно первую литеру директивы размещать в седьмой позиции. Начиная со следующей главы, мы будем систематически использовать «смещения» в изображении программы для подчеркивания ее структуры;
- **позиция 6**: в ней можно поместить любую литеру, чтобы указать, что строка является *продолжением* директивы, начало которой занимают предшествующие строки. Такая строка называется *строкой продолжения;* теоретически можно использовать до 20 последовательных строк (из которых 19 могут быть строками продолжения);
- **позиции с 73 по 80 включительно** предусматриваются для возможной нумерации строк. Эта нумерация факультативна, но некоторые трансляторы при ее наличии контролируют возрастающий порядок номеров строк в программе.

Строка комментария имеет такой формат:

- **позиция** 1: литера *C;*
- **позиции** со второй по 72: произвольный текст, образующий комментарий и игнорируемый транслятором;
- **позиции** с 73 по 80: факультативная нумерация.



II.3.4.3. Некоторые операторы

а) Присваивание

Присваивания в ФОРТРАНе используют знак = , как в следующих примерах:

```
REAL DELTA, A,B,C
INTEGER T(10)
LOGICAL OUEXCL, L1, L2
T(1) = 3
DELTA = B**2–4.*A*C
L = (L1. AND. .NOT. L2) .OR. (.NOT. L1 .AND. L2)
```

Выражение справа от знака = и переменная (или элемент массива), которой присваивается значение этого выражения, должны по стандарту иметь один и тот же тип. Если необходимо оперировать в присваивании с объектами разных типов, надо использовать *функции преобразования типов.* Например, R – вещественная переменная простой точности, а *цел* – целое выражение, тогда записывают, используя функцию преобразования типа *FLOAT:*

R = FLOAT (цел)

чтобы присвоить переменной *R* значение выражения *цел,* представленное как вещественное простой точности. Точно так же, если есть желание присвоить переменной *N* значение вещественного простой точности выражения *вещ* или выражения *DOUBLE PRECISION двовещ,* пишут соответственно

N = INT(вещ)

N = IDINT(двовещ)

Присваивание приводит здесь к потере информации, поскольку происходит переход от дробного значения к целому. *INT IDINT* просто «отрезают» дробную часть (т.е. дают в качестве результата наибольшее целое, абсолютное значение которого меньше или равно рассматриваемому значению вещественного, и обладающее тем же знаком).

Существует 10 функций преобразования типов[1] (см. приложение В). В ряде случаев без них, вообще говоря, можно обойтись: так, *N=вещ* допустимо и эквивалентно *N = INT (вещ).* Однако в той же мере, в какой нужно избегать применения смешанных числовых выражений, рекомендуется *всегда явно называть преобразования типа,* используя функции, предназначенные для этой цели; с одной стороны, действительно, разрешенные комбинации меняются от транслятора к транслятору; с другой стороны, преобразования типов являются операциями, которые могут изменить смысл программы и которые стоят относительно дорого по времени выполнения: поэтому лучше явно указать их присутствие в программе.

Присваивание *строк* ставит особые проблемы. Мы видели в II.3.3.5, что можно объявить переменные целыми, например

INTEGER CARAC, TEX

с намерением дать им значения «литера» или «строка». Мы видели также в II.3.3.1, что существуют константы строкового типа. Следовательно, можно было бы представить, что допустимо писать

CARAC ='U'

Точно так же, если принять соглашение о кодировании более чем одной литеры с помощью целого и предположить для определенности, что работа идет на ИБМ 370 с четырьмя литерами в одном целом (слово), можно было бы представить запись

TEX = 'TACC'

Это было бы слишком просто! В ФОРТРАНе запрещено присваивать

[1] Это число меняется в зависимости от того, насколько широко определено, что такое преобразование типов.

переменной строковую константу. Этого можно избежать, обрабатывая такие константы, как символические константы, инициализируемые предписанием *DATA*.. Например,

```
INTEGER LETTRU, NOM, BLANC
DATA LETTRU /'U'/, NOM /'TACC'/, BLANC/' '/
```

Теперь эти «переменные» можно присваивать переменным:

```
CARAC  = LETTRU
TEX    = NOM
CARAC  = BLANC
```

Отметим, наконец, что если, как выше, (*LETTRU, BLANC*), разместить в целом меньше литер, чем может включать это целое, то предписание *DATA* дополняет в общем случае выделенную область пробелами *справа.* Так, если в дополнение к имеющимся объявлениям объявлено (по–прежнему на ИБМ 370)

```
INTEGER U3BLAN
DATA U3BLAN /'U'/
```

то логическое выражение *LETTRU .EQ. U3BLAN* будет иметь значение *.TRUE.* Но это свойство не абсолютно указано в стандарте, и важно проверять в этом отношении каждый транслятор.

б) Ввод–вывод

Операторы ввода и вывода в ФОРТРАНе более мощные, чем, во многих других языках, но труднее изучаемы. Действительно, в ФОРТРАНе используются одновременно собственно оператор чтения (*READ*) или записи (*WRITE*) и еще одна директива, называемая «форматом». Первый оператор дает список читаемых или записываемых значений, тогда как директива формата описывает форму, принимаемую этими переменными на внешних информационных носителях (число значащих цифр, число пробелов между двумя объектами и т.д.) и преобразования типов, если они необходимы при декодировании. Описание этих операторов выходит за рамки нашей книги, и мы отсылаем читателя к учебникам ФОРТРАНа, указанным в библиографии к этой главе, за полным их описанием. Некоторые приведенные ниже программы содержат оператор ввода и вывода, понимание которых не представит трудностей.

в) *STOP*

Важным оператором является оператор *STOP*, предписывающий останов программы. Важно не путать его с директивой *END* (II.3.4.1), которая является не оператором, а меткой, сообщающей транслятору или человеку, читающему программу, о конце программного модуля.

II.4. Введение в АЛГОЛ W

II.4.1. История

АЛГОЛ 58 и АЛГОЛ 60, прямые предшественники АЛГОЛа W, были результатом работы международной группы экспертов, объединившихся в 1957 г. для создания алгоритмического языка (Algebraic Oriented Language, потом Algorithmic Language), допускающего ясное выражение фундаментальных понятий программирования и свободного от всякого влияния конкретной машины. АЛГОЛ 60 со всей очевидностью достиг этой цели, даже если некоторые аспекты спорны (например, слишком отчетливо высказанная ориентация на численные применения); язык ввел большое число важных понятий и оказал значительное влияние на все последовавшие за ним языки. Следует отметить также важность исходного определяющего документа [Наур 60], который впервые использовал строгий и элегантный метод описания.

Несмотря на все свои достоинства и быструю реализацию трансляторов (с 1960 г.), АЛГОЛ 60 не получил всеобщего распространения, на которое рассчитывали

его создатели. Это имеет ряд причин: сознательно неполный характер определения языка (операторы ввода–вывода и базовый алфавит были оставлены в тени, чтобы не связывать язык с выбором носителя); оппозиция части инженерной среды и некоторое число общественно–научных проблем, которые было бы слишком долго анализировать здесь (см. [Бемер 69]).

Мало распространенный в США, АЛГОЛ имел в Европе преобладающее место в университетских центрах; он является главным языком, используемым в СССР и в Китае, далеко опережая ФОРТРАН. АЛГОЛ в такой же степени, как и алгоритмическая нотация, является главным средством распространения алгоритмов «на бумаге».

После публикации пересмотренного сообщения (1962 г.) и появления новых понятий и новых обозначений (управляющие структуры, структуры данных) у АЛГОЛа 60 стало много последователей. Выделились две основные тенденции. Согласно одной, ориентированной на преподавание, стремились упростить язык, сокращая число базовых понятий ценой возможных потерь в выразительности; эта тенденция привела к языку ПАСКАЛЬ (1970). Согласно другой, пытались обобщить, насколько возможно, фундаментальные понятия, делая возможными любые их комбинации; она привела к АЛГОЛу 68 (1968).

АЛГОЛ W, задуманный Н. Виртом и К. А. Р. Хоаром в 1966 г., стоит на перекрестке путей: концепции АЛГОЛа 60 в нем обобщены в той мере, в какой это можно было сделать простыми средствами; использованы новые идеи, такие, как сложные структуры данных (гл. V), свидетельствующие об эволюции программирования как науки между 1960 и 1966 гг.; но язык остался достаточно простым и достаточно «небольшим», чтобы быть легко освоенным.

АЛГОЛ W – язык, ориентированный на преподавание: существующие трансляторы (для ИБМ 360 и 370, СИМЕНС, а также СII 10070 и ИРИС 80) предлагают пользователю замечательные возможности отладки (ср. VIII.4.2.4 и VII.4.6), но не гарантируют условия эксплуатации и «эффективность», которую имеют право требовать для больших программ профессиональные пользователи. Но, несмотря на эти ограничения, АЛГОЛ W благодаря ясности, с которой он позволяет применять фундаментальные понятия программирования, представляет собой один из самых интересных среди существующих языков.

II.4.2. Значения и типы

В АЛГОЛе W имеют место следующие базовые типы:

- *INTEGER* (целое)
- *REAL* (вещественное)
- *LONG REAL* (вещественное повышенной точности, ср. *DOUBLE PRECISION* в ФОРТРАНе)
- *LOGICAL* (логическое)
- *STRING* (строки; длина строки от 1 до 256 литер)
- *BITS* (битовая конфигурация)
- *COMPLEX* (комплексное) и *LONG COMPLEX* (комплексное повышенной точности)

Литеры обрабатываются как строки (*STRING*), состоящие из одной литеры.

II.4.3. Основные объекты языка

II.4.3.1. Константы

Целая константа записывается в своей обычной десятичной форме с

возможным, непосредственно предшествующим знаком:

0 –3 45678 +5 –273512

Вещественная константа записывается в одной из следующих форм (с возможным, непосредственно предшествующим знаком):

3.14159 0.0314159'02

где апостроф, читаемый «десять в степени...», указывает, что целое, положительное или отрицательное, которое следует за ним (*02* в нашем примере), есть степень десятки; наш последний пример представляется, таким образом, в обычных обозначениях

$$0{,}0314159 \times 10^{2}$$

Целая часть числа или дробная часть (но не обе!) могут быть опущены, если они нулевые. Примеры: *235. .314*

Длинная вещественная константа – это вещественная «короткая» константа, непосредственно сопровождаемая буквой *L* (заметим, что никакая числовая константа не может содержать пробелов).

Комплексная константа – это вещественная константа, за которой непосредственно следует буква *I* (примеры: *3.45I 0.5' – 10I*).

Длинная комплексная константа – это длинная вещественная константа, непосредственно сопровождаемая буквой *I* (пример: *3.45LI*). Заметим, что так представляются только чисто мнимые числа; комплексное число в обычном смысле будет рассматриваться как выражение, например *3.25+47.2I*.

Логические константы – это *TRUE* или *FALSE* («истина» или «ложь»).

Строковая константа это последовательность из литер от 1 до 256, заключенная в кавычки , например

"ЭТО СТРОКА"

Если одна из литер строки есть литера «кавычка»[1], она должна быть повторена в описании константы. Так, строковая константа

"АЛЛА ТАРАСОВА ИГРАЛА В ""АННЕ КАРЕНИНОЙ"""

образована литерами

АЛЛА ТАРАСОВА ИГРАЛА В "АННЕ КАРЕНИНОЙ"

II.4.3.2. Переменные

Идентификаторы АЛГОЛа W образуются произвольным количеством литер, из которых первая должна быть буквой, а последующие могут быть буквами, цифрами или литерой «подчеркивания». Примеры:

A B1 IDENTIFICATEUR A_ BEFORE_B

Всякая переменная должна быть объявлена. Примеры объявлений[2]:

INTEGER A
LOGICAL U,V,W,X,Y
LONG REAL C_D_E

Для объявления переменной типа СТРОКА нужно уточнить длину (фиксированную) етрок, которые переменная может принимать в качестве значения. Примеры:

STRING (1) TEXTE_1
STRING (27) TEXTE_2, TEXTE_3, TEXTE_4

TEXTE_2, *TEXTE_3*, *TEXTE_4* могут принимать своими значениями строки из 27 литер; *TEXTE_1* – это цепочка из одной литеры. Можно написать просто *STRING Z,* что

[1] Речь идет о литере «кавычка», а не о повторных апострофах.

[2] В языках типа АЛГОЛа объявления традиционно называются описаниями. Мы сохраняем термин «объявление» для единства терминологии. – *Прим. перев.*

эквивалентно *STRING (16)Z*.

Из переменной *T* типа СТРОКА можно образовывать подстроки: если написать *T(e|n),* где *e* – целое выражение, положительное или нулевое, а *n* – целая положительная константа, это будет обозначать строку длиной *n,* полученную из *T* выделением *n* литер, начиная с *e+1–й* (или, если угодно, начиная с *e*–й позиции при соглашении, что первая литера занимает нулевую позицию).

Такая подстрока может использоваться как переменная, т.е. либо для того, чтобы получить значение *n* смежных литер из *T,* либо чтобы изменить значение этих *n* литер. Так, если объявлено *STRING (8) T*, а *T* имеет значение *"ПЕРЕМЕНА",* то *T(4|4)* имеет своим значением *"МЕНА"* Если с помощью оператора присваивания (см. ниже) *T(4|4)* получает значение *"ДАЧА",* то значением *T* будет *"ПЕРЕДАЧА"*..

II.4.3.3. Массивы

Массив трех измерений (например) с границами $[m_1\ M_1]$, $[m_2,\ M_2]$, $[m_3,\ M_3]$ объявляется с помощью

INTEGER ARRAY A (m1 :: Ml, m2 :: M2, m3 :: M3)

INTEGER может быть заменено *REAL LONG REAL STRING(n)* и т.д.

m_1 M_1 и т.д. могут быть:

- либо целыми константами, положительными, отрицательными или нулевыми, для которых $m_i \le M_i$
- либо переменными или выражениями; в этом случае все переменные, участвующие в выражениях границ, должны быть объявлены в блоке, в котором объявлен массив *A* (см. III.5.1); эти переменные должны быть инициализированы динамически перед входом в этот последний блок. При каждом начале выполнения блока должно сохраняться $m_i \le M_i$.

Несколько массивов одинаковой размерности и с одинаковыми границами могут быть объявлены вместе, как в следующем примере:

STRING(7) ARRAY T1, T2, T3 (–3::24)

Язык не определяет способ, которым элементы массива размещаются в памяти.

II.4.3.4. Выражения

а) *Арифметические выражения*

Они формируются из констант и переменных с помощью следующих знаков операций:

+ – * /	(сложение, вычитание, умножение, деление)
DIV REM	(целое деление и целый остаток)
**	(возведение в степень)
LONG SHORT ABS	(унарные операторы обращения)

Операнды у +, –,*, / имеют любые арифметические типы (*INTEGER*, *REAL*, *LONG REAL*, *COMPLEX*, *LONG COMPLEX*) Операнды у *DIV* и *REM* должны быть целыми. Первый операнд ** имеет любой арифметический тип; второй (показатель) – обязательно целый. *LONG x,* где *x* типа *INTEGER, REAL* (или *COMPLEX*), это представление $_x$ как *LONG REAL* (или *LONG COMPLEX*). *SHORT x,* где *x* типа *LONG REAL* (или *LONG COMPLEX*), это приближение *x* с помощью *REAL* (или *COMPLEX*). *ABS x,* где *x* имеет любой арифметический тип, относится к тому же типу, что и *x,* и в качестве значения имеет абсолютное значение *x.*

В таблицах приложения Г можно найти результирующие типы комбинаций возможных типов для операций +, –, *, / и **.

Имеют место три следующих общих принципа:

1) Результат имеет «точность», необходимую для того, чтобы дать ему *смысл*

во всех возможных случаях. Так,

*a**n*

(где *n* должно быть целым) имеет тип *LONG REAL,* когда *a* имеет тип *INTEGER, REAL* или *LONG REAL*. Действительно, даже для *a* целого a^n не может быть надлежащим образом представлено целым при отрицательных *n*. Если необходимо представить целым, например, a^3, записывают *a*a*a*.Точно так же

i/j

имеет тип *LONG REAL* для *i* и *j* типа *INTEGER;* например, *8/3* дает *2.666...6L*. Чтобы получить целые частное и остаток при делении *i* на *j,* записывают соответственно

i DIV j («целое частное»)

и *i REM j* («целый остаток»)

например, *28 DIV 3* дает *9*, а *28 REM 3* означает *1*.

2) Сохраняется степень точности, обеспечивающая законность вычислений. Так,

r + lr

где *r* имеет тип *REAL,* а *lr* – тип *LONG REAL* будет иметь своим типом *REAL:* нельзя, действительно, безосновательно увеличивать точность *r* до *LONG REAL*. Можно получить в результате *LONG REAL,* записав *LONG r + lr*. *LONG* имеет более высокий приоритет по сравнению с + – (см. г) ниже); это выражение отличается от *LONG (r + lr)*; оно имеет большую точность, поскольку вычисление последнего выражения начинается «обрезанием» *lr* для выполнения сложения, а затем результат снова «растягивается».

3) Когда по крайней мере один операнд комплексный («короткий» или «длинный»), результат тоже будет комплексным.

б) *Выражения «строкового» типа*

Строковые выражения могут быть константами или переменными строкового типа. Напомним, что последняя категория содержит «подстроки»; так, если *T* есть переменная, объявленная *STRING (10)* то *T(7|2)* представляет собой подстроку *T* длиной 2, сформированную из ее восьмой и девятой литер. Пусть *T1* объявлено

STRING ARRAY T1 (0::5)

Тогда подстрока, соответствующая третьему элементу *T1*, обозначается *T1(3)(7|2)*.

в) *Логические выражения*

Логические выражения содержат прежде всего логические константы *TRUE* и *FALSE* и переменные, объявленные *LOGICAL*.

Второй тип логических выражений образуется из арифметических выражений и операторов отношений

= < > >= <= ¬ ¬=

последний из этих операторов означает «отличается от». Так, *A < B* истинно (значит, *TRUE*), если и только если *A* строго меньше *B*. Можно сравнивать между собой с помощью операторов отношений числовые значения типов *INTEGER, REAL, LONG REAL;* транслятор выполняет необходимое преобразование типов, если типы сравниваемых значений различны.

Такие же операторы позволяют в равной мере сравнивать две строки; в этом случае операторы отношений < , > , <=, >= используют алфавитный порядок. Этот алфавитный порядок официально определен только для букв; для других литер он зависит от конкретной машины и используемых кодов. Если две строки имеют разные длины, то сравнение производится после того, как более короткая строка будет дополнена пробелами справа до длины более длинной строки: пробел считается литерой, предшествующей всем буквам в алфавитном порядке.

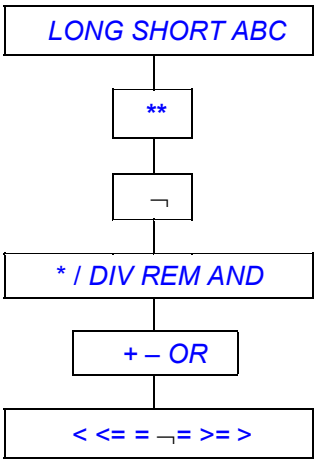
С помощью произвольных логических выражений el_1 и el_2 можно, кроме того, образовывать такие логические выражения:

el_1 *AND* el_2	(«конъюнкция») истинно, если и только если el_1 и el_2 истинны;
el_1 *OR* el_2	(«дизъюнкция») истинно, если и только если по крайней мере одно из выражений el_1, el_2 истинно;
$\neg\, el_1$	(«отрицание») истинно, если и только если el_1 ложно;
$el_1 = el_2$	(равенство, или «идентичность») истинно, если и только если el_1 и el_2 оба истинны или оба ложны;
$el_1 \neg= el_2$	(«или исключающее») истинно, если и только если $el_1 = el_2$ ложно.

Заметим, что приоритет *AND* выше, чем *OR*. *A AND B OR C AND B,* означает *(A AND B) OR (C AND D)*. В сомнительных случаях расставляют скобки.

г) *Приоритет операций*

Иерархия операций АЛГОЛа W по убыванию приоритетов показана на схеме



В этой схеме появляется аномалия: приоритет *AND* и *OR* выше, чем у операторов отношения, т.е. нельзя опускать скобки в примерах

(A = B) AND (C > D)

или *(U + V ¬= C + D) OR (H >= F)*

«Унарные» + и – имеют тот же приоритет, что и их бинарные эквиваленты, и приравнены к ним в вышеприведенной схеме.

д) *Условные выражения*

В АЛГОЛе W можно образовывать условные выражения, значения которых зависят от значений логических выражений. Положим (для простоты), что e_1 и e_2 имеют одинаковый тип и el – логическое выражение. Тогда выражение

IF el THEN e_1 *ELSE* e_2

означает e_1, если *el* истинно, и e_2 в противном случае. Его тип тот же, что и у e_1 и e_2.

Пусть, например, имеют место объявления

INTEGER X,Y; LOGICAL U,V

Тогда можно написать условное выражение

IF X > Y THEN X ELSE Y

(Это выражение имеет целое значение – максимум значений *X* и *Y*,

IF U THEN V ELSE FALSE имеет то же значение, что *U AND V*

IF U THEN TRUE ELSE V имеет то же значение, что *U OR V*

IF U THEN FALSE ELSE TRUE имеет то же значение, что *¬U*

(три последних выражения имеют тип *LOGICAL*).

II.4.4. Программы, операторы

II.4.4.1. Физическая форма программы

Формат программы АЛГОЛа W свободный, т.е. размещение текста программы на странице не оказывает влияния на программу при следующих ограничениях:

- Программа рассматривается как последовательность 80–литерных строк, каждая из которых соответствует одной перфокарте. Принимаются во внимание только первые 72 литеры каждой строки; разделение двух последовательных строк специально не обозначается.
- Литеры, составляющие число (пример: *–293.27' –3L*), идентификатор или зарезервированное слово (как *DIV, LONG* и т.д.) должны быть написаны последовательно без пробелов между ними.
- Если за идентификатором, зарезервированным словом или константой следует идентификатор, зарезервированное слово или константа, то они должны разделяться по крайней мере одним пробелом. Это не обязательно, если между элементами таких пар располагаются одна или более неарифметических литер, таких, как +, –, *(,)*, и т.д.
- Любое число пробелов всегда эквивалентно единственному пробелу (исключение: в константе типа СТРОКА, где пробел – это одна из литер строки). Это позволяет разрядить расположение текста программы на странице, чтобы улучшить читаемость программы, и мы будем пользоваться этим систематически со следующей главы, чтобы подчеркнуть «блочную структуру» АЛГОЛа.
- Всякая последовательность литер, заключенная между зарезервированным словом *COMMENT* (комментарий), сопровождаемым пробелом или знаком операции с одной стороны и точкой с запятой с другой стороны, рассматривается как комментарий, т.е. не анализируется транслятором. Комментарий может появиться в любом месте, где разрешен «основной объект», т.е. всюду, кроме середины числа, идентификатора и т.д.

Программа АЛГОЛа W начинается зарезервированным словом *BEGIN* и заканчивается зарезервированным словом *END,* сопровождаемым точкой: *END.*.

Между *BEGIN* и *END* содержатся сначала **объявления,** затем **операторы,** отделяемые друг от друга точкой с запятой.

II.4.4.2. Некоторые операторы

а) *Присваивание*

Оператор присваивания записывается в АЛГОЛе W как

пер := выр

:= («двоеточие», сопровождаемое знаком «равенства», читается «принимает значение») есть знак оператора присваивания, который не следует путать со знаком оператора отношения =; *пер* – *это* объявленная переменная или разрешенный элемент объявленного массива; *выр* – это выражение.

Обычно *пер* и *выр* должны быть одного типа. Однако:

1. переменной типа СТРОКА, объявленной с некоторой длиной, можно присваивать выражение типа СТРОКА с более коротким значением: строка при этом дополняется пробелами справа;

2. переменной числового типа (целое, вещественное длинное и простое, комплексное длинное и простое) можно присваивать выражения менее «точных» типов

(считая, что *LONG REAL* и *LONG COMPLEX* более «точны», чем *REAL* и *COMPLEX,* а последние в свою очередь более точны, чем *INTEGER*) при условии, что переменная комплексная («короткая» или «длинная»), если таковым является выражение. Так, при объявлениях

```
INTEGER I; REAL X; LONG REAL A;
STRING (5) U; STRING (3) T;
```

следующие присваивания законны:

```
I := 3;
X := I COMMENT:X VAUDRA 3.0;;
T : = "AAA";
T := "BB" COMMENT : T VAUDRA "BB";;
U := T COMMENT :U VAUDRA "BB " ;;
A := X COMMENT A VAUDRA 3.0 L;;
U(1|3) := "CC" COMMENT U VAUDRA "BCC ";
```

Можно присвоить одно и то же значение нескольким переменным с помощью многократного присваивания вида

пер1 := пер2 := ...:= перп : = выражение

б) *Ввод–вывод*

Оператор *READON(A, B, C, D, E)*, где *A, B, C, D, E* – переменные или элементы массива произвольных типов и в любом количестве, предписывает чтение соответствующих констант с карт и присваивает их значения соответственно переменным. Тип читаемых констант должен быть таким, который можно присвоить *A, B, C, D, E* соответственно с соблюдением рассмотренных выше правил.

Оператор *WR1TEON(E1, E2, E3, E4),* где *E1, E2, E3, E4* – выражения произвольных типов и в любом количестве, предписывает отпечатать значения этих выражений в указанном порядке и соответствующем им формате. Так, если выполняется оператор

*WRITEON(I,"В КВАДРАТЕ РАВНО", I*I)*

и если значение целой переменной *I* равно 259, то программа напечатает

259 В КВАДРАТЕ РАВНО 67081

II.5. Введение в ПЛ/1

II.5.1. История

Сокращение «ПЛ/1» – "Programming Language number 1", или «Язык программирования № 1», – достаточно хорошо отражает претензии создателей этого языка: речь шла «всего навсего» о том, чтобы заменить ФОРТРАН, КОБОЛ, АЛГОЛ и все другие языки, существовавшие к началу шестидесятых годов, новым языком (первым предложенным сокращением было NPL – «New Programming Language», «Новый язык программирования»), который объединил бы все их достоинства и не испытывал бы никакого из различных их ограничений.

ПЛ/1, который увидел свет в 1963–1964 гг., явился результатом совместных усилий ИБМ и двух пользовательских ассоциаций ИБМ "SHARE" и "GUIDE". Хотя с тех пор ПЛ/1 добился значительного распространения, справедливо отметить, что начальные цели – вытеснить АЛГОЛ, ФОРТРАН и КОБОЛ – не были достигнуты. Это объясняется рядом причин: по сравнению с ФОРТРАНом язык ПЛ/1 невыгодно отличается своей сложностью, а его «оптимизирующие» трансляторы задыхаются, пытаясь достичь совершенств фортрановских трансляторов; многие черты были заимствованы в АЛГОЛе, но элегантность и простота этого языка были потеряны;

сравниваемый с КОБОЛом, наконец, ПЛ/1 страдает от того, что долго был доступен только на машинах ИБМ и во всяком случае далек от той же степени стандартизации.

Основные характеристики языка – его выразительная способность и сложность. В самом деле, ПЛ/1 позволяет обрабатывать задачи в самых разных прикладных областях. Но язык представляется порой скорее разношерстным, нежели универсальным (его противники характеризуют его как набор рецептов, взятых из ФОРТРАНа, АЛГОЛа 60 и КОБОЛа); он трудоемок для систематического изучения; наконец, его сложность затрудняет написание эффективных трансляторов. Несмотря на эти недостатки, ПЛ/1 занимает солидную часть «рынка», особенно в экономических приложениях. Судьба языка в долгосрочной перспективе остается, однако, неопределенной.

Описывая ПЛ/1, мы ограничимся теми свойствами языка, которые реально полезны в повседневной работе программиста. Учитывая большой объем языка, почти все программисты сами ограничиваются его некоторым подмножеством;

II.5.2. Значения и типы

ПЛ/1 предлагает более широкий набор основных типов, чем другие языки; к сожалению, определение и использование этих типов непосредственно связано с особенностями машин ИБМ.

В ПЛ/1 допускаются следующие типы:

- строка (*CHARACTER*)
- битовая строка (*BIT*). Логическое значение обрабатывается в форме битовой строки *BIT* длиной 1.
- *PICTURE* (шаблон), позволяющий обрабатывать форматы ввода и вывода, и *FILE* (файл), разрешающий работу с файлами. К таким типам мы не будем обращаться в этой книге
- два особых типа, о которых у нас будет повод еще раз поговорить в гл. III и IV: *LABEL* (метка) и *POINTER* (указатель)
- числовой тип, который объединяет предполагаемые другими языками типы «целый», «вещественный» («длинный» или нет) и «комплексный», но различает их благодаря атрибутам с многочисленными возможными комбинациями. Существует четыре числовых атрибута:

а) атрибут основания указывает, должно ли значение рассматриваться в двоичной системе счисления (*BINARY*) или в десятичной (*DECIMAL*);

б) атрибут вида, указывающий, является ли число вещественным (*REAL*) или комплексным (*COMPLEX*);

в) атрибут формы представления указывает, имеет ли число представление с фиксированной запятой (*FIXED* – это, в частности, ситуация обычного представления целых), или оно отнесено к некоторому «масштабу», который является степенью основания 2 или 10; такое значение называется «с плавающей запятой» (*FLOAT*), что означает использование пары (мантисса, порядок) в представлении числа: см. II.1.1.5;

г) атрибут **разрядности** указывает число значащих цифр, входящих в представление числового значения:

– для числа с *фиксированной запятой* разрядность определяется парой [p, q] двух целых; *p – это* общее количество цифр, изображающих число (двоичных или десятичных); *q* – количество цифр справа от запятой, *p* ограничено в каждой системе; на ИБМ 360 или 370 максимальные значения для вещественных чисел:

$p = 31$ в двоичной системе; $p = 16$ в десятичной.

II.5.3. Основные объекты языка

II.5.3.1. Константы

а) *Строковая константа* (*CHARACTER*) записывается между двумя апострофами. Если такая строка содержит литеру «апостроф», эта литера изображается двумя апострофами. Примеры:

'ABRACADABRA'

'ОБ'ЯВЛЕНИЕ ОБ ОБ'ЕЗДЕ' (константа типа СТРОКА, образованная из литер ОБ'ЯВЛЕНИЕ ОБ ОБ'ЕЗДЕ).

б) *Константа «битовая строка»* содержит цепочку знаков *0* и *1*, перед которой стоит апостроф и сопровождаемую апострофом и буквой *B* (для «битов»):

'100101001'B

'1'B и *'0'B* (обычные представления логических констант **истина** и **ложь**).

в) *Числовые константы* записывают по–разному в зависимости от значений их атрибутов. Комбинации атрибутов допускают большое число возможностей.

В описание числовой константы могут входить цифры от *0* до *9*, знаки + и – и буквы:

E (для того, чтобы указать, что последующие цифры представляют показатель: число имеет, следовательно, атрибут формы представления *FLOAT*).

B (для того, чтобы указать, что число имеет атрибут двоичного основания. Если буквы *B* нет, то система счисления десятичная).

I (для того, чтобы указать, что атрибут вида «комплексный». Если *I* отсутствует, то вид «вещественный». Заметьте, что используют *I* от *IMAGINARY,* а не *C*). Если в изображении числа есть буква *B* или буква *I*, то она размещается в конце представления числа (относительный порядок *B* и *I* произволен, *BI* или *IB,* если присутствуют обе литеры).

Таким образом, числовая константа имеет общий вид

(1) $$\pm c_1c_2...c_n{-}c_{n+1}c_{n+2}...c_{n+m}E\pm d_1d_2BI$$

где $c_1 ..., c_{n+m}$ – двоичные или десятичные цифры, d_1 и d_2 – десятичные цифры; *BI* может быть заменено на *IB;* некоторые элементы могут отсутствовать (действительно, достаточно одной цифры c_1 чтобы сформировать числовую константу).

Различные атрибуты выводятся из формы константы (или, если угодно, определяют ее):

- атрибут основания *двоичный* при наличии *B, десятичный* в противном случае. Если константа *двоичная,* цифры $c_1, ..., c_{n+m}$ должны, разумеется, быть цифрами *0* или *1*; напротив, d_1 и d_2, представляющие показатель, всегда десятичные цифры (!);
- атрибут вида является *комплексным,* если присутствует буква *I,* и *вещественным* в противном случае;

атрибут формы представления есть *плавающая запятая* (*FLOAT*), если представлена часть $E\pm d_1d_2$, и фиксированная запятая, если она отсутствует. В противном случае $\pm d_1d_2$ представляет степень десятки, например:

+3.25E + 03, т.е. $3{,}25 \times 10^3$

–1010.0E–16B, т.е. -10^{-16} (действительно, двоичное число 1010 есть десятичное 10).

В $E \pm d_1d_2$ можно опустить знак показателя, если он +, а также d_1, если $d_1 = 0$: *+3.25E3,*

- атрибут разрядности определяет число значащих цифр. Для числа с фиксированной запятой он задается парой [*n + m,m*]; так, *2884.0* и *1010.1BI* имеют разрядность [5,1]: пять цифр, из которых одна после запятой.

Для числа с плавающей запятой атрибут разрядности задается количеством цифр, указанных перед *E,* т.е. *n + m: 874.5E–12* и *0.011E27* имеют разрядность 4. Заметьте, что *00.011E27* имело бы разрядность 5(!).

Обычно программист почти никогда не занимается атрибутом разрядности констант. К сожалению, он обязан принимать меры предосторожности: как мы видели в II.5.3.4.г, *0.011E27* и *00.011E27* могут вести себя по–разному в выражениях!

Отметим, наконец, что в общей форме (1) знак + в начале числа может быть опущен, так же как и цифры перед десятичной точкой и после нее, если они нулевые. Если число не имеет дробной части, можно опустить десятичную точку.

В заключение рассказа об этих сложных правилах дадим несколько иллюстрирующих примеров.

Константа	Основание	Форма представления	Вид	Разрядность	Примечания
1983	десятичное	фиксированная	вещест-венный	[4,0]	целая константа
– 76.54	десятичное	фиксированная	вещест-венный	[4,2]	
11100B	двоичное	фиксированная	вещест-венный	[5,0]	значение 28
–101.01B	двоичное	фиксированная	вещест-венный	[5Д]	значение –5,25
6.02E23	десятичное	плавающая	вещест-венный	3	
101.1E–3B	двоичное	плавающая	вещест-венный	4	значение $5{,}5 \times 10^3$
1011E–1B	двоичное	плавающая	вещест-венный	4	значение 0,55; показатель десятичный
101.1E–1BI	двоичное	фиксированная	комплекс-ный	4	значение 0,55i
–54I	десятичное	фиксированная	комплекс-ный	[2,0]	разные разрядности
–54.00I	десятичное	фиксированная	комплекс-ный	[4,2]	
–054.000I	десятичное	фиксированная	комплекс-ный	[6,3]	
954.72E26B					недопустимо

II.5.3.2. Переменные

а) *Идентификаторы*

Идентификатор в ПЛ/1 это последовательность от 1 до 31 литер, первая из которых должна быть обязательно буквой, а остальные – буквами, цифрами или литерой подчеркивания. Можно использовать также несколько специальных литер, рассматриваемых в качестве алфавитных, но они не являются ни в коей степени объектом стандартизации языка, что не дает возможности рекомендовать их использование.

б) *Общая форма объявлений*

Объявление имеет общую форму

DECLARE ид атрибут–1 атрибут–2 ... атрибут–n;

где *ид* – идентификатор, *атрибут–1, атрибут–2, ..., атрибут–n* –имена «атрибутов», которые служат для определения типа, как будет показано ниже; точка с запятой – неотъемлемая часть объявления[1]. Пример:

DECLARE T FLOAT DECIMAL;

С помощью скобок и запятых можно сгруппировать объявления нескольких переменных, характеризуемых одними и теми же атрибутами:

DECLARE (ид–1, ид–2, ..., ид–n) атрибут–1, ..., атрибут–n;

Пример:

DECLARE (S,P,Q,R) FLOAT DECIMAL;

Точно так же с помощью запятых можно объединить объявления, соответствующие различным атрибутам. Например,

DECLARE (S,N,C,F) FLOAT DECIMAL
V FIXED BINARY,
(R,A,T,P) CHAR (30);

в) *Переменные строкового типа*

Переменные строкового типа могут быть объявлены имеющими фиксированную длину *l* или переменную во время выполнения длину с заданным максимумом *m*.

В первом случае атрибутом будет *CHARACTER(l),* во втором случае будут указаны два атрибута *CHARACTER(m) VARYING*.

Примеры объявлений:

DECLARE (TEX_1, TEX_2) CHARACTER(20) VARYING;
DECLARE CARTE CHARACTER(80);

Строки переменной длины разрешают гибкое программирование. Строки фиксированной длины делают возможным более быстрый доступ ценой возможной потери места (если максимум длины, переменной во время выполнения, неизвестен, то можно привести такую ситуацию к рассмотренной, взяв *l* заведомо слишком большим).

г) *переменные типа BIT*

Возможными атрибутами являются *BIT(l),* где *l* – фиксированная длина соответствующей цепочки битов, и *BIT(m) VARYING,* где *m* – максимум длины во время выполнения. Примеры;

DECLARE ENSEMBLE BIT(100) VARYING,
M BIT (32),
(ДА, НЕТ) BIT (1);

Последние две переменные имеют логический тип.

д) *Числовые переменные*

Как мы видели, есть четыре атрибута:

- основание, задаваемое словами *BINARY* (двоичное) или *DECIMAL* (десятичное);

- вид, задаваемый словами *REAL* (вещественный) или *COMPLEX* (комплексный);

- форма представления, задаваемая словами *FLOAT* (плавающая) или *FIXED* (фиксированная);

- разрядность, представляемая числом *n* значащих цифр, если форма представления *FLOAT,* и парой (n_1, n_2), задающей общее количество цифр и количество

[1] Большинство ключевых слов ПЛ/1 имеет «сокращения»; так, можно писать *DCL* вместо *DECLARE* Мы будем использовать, как правило, полную форму.

цифр после запятой, если форма представления *FIXED*. Указания разрядности и формы представления связывают, записывая *FLOAT(n)* или *FIXED(n_1, n_2)*. Напомним, что существуют максимальные значения для *n* и n_1.

Таким образом, полное объявление на ИБМ 360 или 370 для объектов *N, D* и *C*, эквивалентное тому, что известно в ФОРТРАНе под именами соответственно *INTEGER, DOUBLE PRECISION* и *COMPLEX (INTEGER, LONG REAL* и *COMPLEX* в АЛГОЛе W), имеет вид

```
DECLARE N BINARY REAL FIXED (31, 0),
        D BINARY REAL FLOAT (53),
        C BINARY COMPLEX FLOAT (21);
```

Чтобы избавить программиста от постоянного описания всех атрибутов, зафиксированы атрибуты *по умолчанию,* которые выбирает транслятор, когда некоторые указания отсутствуют. Например, если нет атрибута вида, неявно принимается *REAL;* поэтому атрибут вида задают только в том частном случае, когда он *COMPLEX*.

К сожалению, экстравагантность других альтернатив по умолчанию усложняет их систематическое использование. Если отсутствует атрибут формы представления, в качестве этого атрибута принимается *FLOAT,* если присутствует один из атрибутов *BINARY, DECIMAL, REAL, COMPLEX;* если отсутствует атрибут основания, он принимается *DECIMAL* при наличии одного из атрибутов *FIXED, FLOAT, REAL, COMPLEX*. В отсутствие всякого объявления транслятор не выдает сообщения об ошибке, а в лучших традициях ФОРТРАНа связывает атрибуты *BINARY REAL FIXED (15,0)* с каждым идентификатором, начинающимся с *I, J, K, L, M* или *N,* и атрибуты *DECIMAL REAL FLOAT(6)* со всеми другими.

Поэтому рекомендуется систематически указывать все атрибуты, за исключением атрибута вида (и атрибута разрядности в наиболее распространенных случаях). Вот наиболее используемые на ИБМ 360/370 числовые «типы»:

BINARY FIXED (31,0) (целые)

BINARY FIXED (15,0)

или просто *BINARY FIXED* (целые, заключенные между —32768 и +32767)

BINARY FLOAT (53) (вещественные «удвоенной точности»)

е) атрибут *INITIAL*

Можно задавать начальное значение переменным любого типа, включая в объявление *DECLARE* атрибут

INITIAL (v)

где *v* – константа; переменная «начнет» участвовать в выполнении программы со значением *v*. Пример:

```
DECLARE PI DECIMAL FLOAT (9) INITIAL (3.14159265E0),
        E DECIMAL FLOAT (9) INITIAL (2.71828183E0),
        AVOGADRO DECIMAL FLOAT (3) INITIAL (6.02E23),
        MARIGNAN DECIMAL FIXED (4,0) INITIAL (1515);
```

Так же как и для директивы *DATA* в ФОРТРАНе, рекомендуется сохранять атрибут *INITIAL* для объявлений «символических констант» (II.1.2.1), за исключением той инициализации переменных, которая находится в ведении операторов присваивания.

> Риск путаницы здесь меньше, чем в ФОРТРАНе, так как переменные, управляемые «динамическим распределением» *AUTOMATIC* (IV.6.5), вновь инициализируются при каждом выполнении «блока», которому они принадлежат, если они обладают атрибутом *INITIAL* Ср. также проблему инициализации остаточных переменных *STATIC*. Эти проблемы обсуждаются в IV.6.5.

II.5.3.3. Массивы

Идентификатор, появляющийся в *DECLARE,* представляет массив, а не переменную, если его имя непосредственно сопровождается заключенным в скобки списком пар границ. Например, в

DECLARE T(1 :10, –5:11) CHARACTER(20);

T объявлен двумерным массивом, элементы которого – строки фиксированной длины 20. Границами служат 1 и 10 для первого измерения, –5 и 11 для второго измерения. Когда в паре границ *a:b* нижняя граница *a* равна 1, то *a:* можно отпустить; здесь, следовательно, можно было бы объявить

DECLARE T(10, –5:11) CHARACTER(20);

В этом массиве 170 элементов, так как 0 является разрешенным индексом для второго измерения.

Для массива можно использовать атрибут *INITIAL,* задавая список начальных значений, присвоенных элементам массива. Для этого можно использовать, если это необходимо, *коэффициент повторения;* например, записывают

DECLARE T(10, –5:11) FIXED DECIMAL(7) INITIAL ((170)0);

для того, чтобы инициализировать нулями весь массив *T*. Если требуется установить единицу в 17 элементах «строки» и нуль в 153 других элементах, надо писать *INITIAL ((17)1, (153) 0)*. В самом деле, в языке принято, что элементы массива более чем одного измерения размещаются в памяти по строкам (соглашение, обратное по отношению к принятому в ФОРТРАНе):

T(1,–5),T(1,–4),...,T(1,11), T(2,–5),...,T(2,11), ..., T(10,–5),... ,T(10,11)

1 строка 2 строка 10 строка

и в таком порядке им присваиваются значения, специфицированные атрибутом *INITIAL*

Можно объявлять и другие группирования данных, отличные от массивов; речь идет об иерархических структурах, роль которых изучается в гл. V, посвященной структурам данных.

II.5.3.4. Выражения

Выражения ПЛ/1 позволяют комбинировать константы, переменные, встроенные функции (см. ниже), массивы, «структуры» (гл. V) с помощью различных операций.

а) *Выражения строкового типа* (*CHARACTER*) используют операцию конкатенации | |. Так,

'АЙ' | | БОЛИТ равно *'АЙБОЛИТ'*

Если объявлено

DECLARE (T1, T2) CHARACTER (4);

и если строка *T1* есть *'ПАПА',* а *T2* – *'ГЕНА',* то T1| |T2 – это строка длиной 8, которая равна *'ПАПАГЕНА'*.

Можно таким же образом образовывать выражения строкового типа с помощью встроенной функции *SUBSTR,* определяемой ниже в п. е.

б) *Выражения типа BIT* используют операции обработки битовых строк:

& (логическое **и**)

| (логическое **или**)

¬ (**не** – логическое дополнение)

Эти операции работают поразрядно (бит с битом) в операндах. Так,

¬*'0110'B* дает *'0100'B*

'0110' B & '1100' B равно *'0100'B*

'0110'B | '1100' B равно *'1110'B*

Частный случай выражений *BIT* образуется логическими выражениями или в ПЛ/1 – *BIT(1);* их можно получить, исходя из числовых операндов и операторов отношений. Значением этих выражений является *'1'B* для **истины** и *'0'B* для **лжи**. Используются следующие операторы отношений:

<<==> = >

¬= (отличен от) ¬< (не меньше) ¬> (не больше)

(¬< это синоним >=, а ¬> синоним <=).

Примеры:

3.7 < 9.2 равно *'1'B* **(истина)** *79 ¬=79* равно *'0'B* **(ложь)**

Операторы отношений, устанавливающие порядок (<, > <=, ¬<, ¬>), могут применяться и к операндам строкового типа. В этом случае используемый порядок есть обычный алфавитный порядок; литера «пробел» предшествует всем буквам, за ними следуют цифры. На конкретной машине задан алфавитный порядок также и для других литер, но без всякой гарантии совместимости с другими реализациями.

Сравнение двух строк разной длины делается дополнением более короткой строки пробелами справа. Примеры:

'PRAXITELE' < 'RASTRELLI' '1'B (**истина**)

'АЛГОЛ 68' >= 'ПЛ/1' '0'B (**ложь**)

в) *Арифметические выражения* используют операции + и –(унарные и бинарные), *, / и ** (возведение в степень).

г) *Проблема преобразования типов*

ПЛ/1 исключительно гибок в своих возможностях комбинировать операнды различных типов в одном выражении. Действительно, разрешено почти все.

Рассмотрим следующие объявления:

```
DECLARE (T1, T2) CHARACTER (20) VARYING,
        (X, Y) BINARY FIXED (15,0);
```

Выражение

(1) *((T1 < T2)*X) + (T1 >= T2)*Y*

имеет значение *X,* если *T1* предшествует *T2* в алфавитном порядке, и *Y* в противном случае. В самом деле, *T1 < T2* имеет результатом *'1'B* или *'0'B;* эта битовая строка длиной 1 преобразуется в *1* или *0* (числовые), чтобы быть умноженной на *X;* то же самое можно сказать о результате *T1 >= T2,* умножаемом на *X.*

Возможны еще более сложные преобразования типов: из числовых (например, *DECIMAL COMPLEX FLOAT(5)*) в строковый, из *BINARY REAL FIXED (4,2)* в *BIT (2)* и т.п. Выражение

(2) *'1'B<2 < '3'*

которое заставляет участвовать *B1T(1),* операнд *DECIMAL FIXED(l)* и операнд *CHARACTER(1)* имеет своим значением *'1'B* (действительно, чтобы его вычислить, сравнивают *'1'B* и 2, преобразованные в *BINARY FIXED,* чтобы дать *1B* и *10B;* результатом становится истина, т.е. *'1'B,* которая преобразуется в строку, чтобы дать *'1',* теперь *'1'* сравнивается с *'3'*).

Мы приводим примеры, разумеется, не в качестве образца, а скорее для того, чтобы показать к каким экстравагантностям приводит примиренчество в языках программирования. Приведенный выше пример (1) представляет собой «условное выражение» (ср. II.2), как мы видели в АЛГОЛе W, и которое будет вычисляться в ПЛ/1 «условным оператором» (III.5.3); поскольку в этом примере фактически необходимы вычисления, становится целесообразным сделать очевидными, насколько возможно,

последовательные преобразования типов. ПЛ/1, как и другие языки, имеет функции преобразования типов (список в приложении Е), которые всегда можно использовать явным образом.

Преобразования типов, к сожалению, украдкой появляются в наиболее частых числовых операциях и могут стать причиной серьезных трудностей, если им не уделить особого внимания. Проблема возникает из–за того, что в ПЛ/1 имеется единственный числовой тип, формируемый из многочисленных «подтипов», которые определяются атрибутами, в частности атрибутом разрядности. В арифметическом выражении атрибуты результата определяются атрибутами операндов; поэтому часто приходится задавать вопрос о разрядности операндов, в частности когда речь идет о константах, которые могут иметь различные атрибуты в зависимости от способа, которым они описаны.

Схематически правила для вида *REAL* таковы (за исключением операции **, которая подчиняется более сложным условиям): – когда в выражении участвуют операнды *FIXED* с разрядностями $[p_1\ q_1]$ и $[p_2, q_2]$, разрядность результата определяется как

$[(1 + \max(p_1 - q_1, p_2 - q_2) + \max(q_1, q_2)), \max(q_1, q_2)]$ для + и –

$[p_1 + p_2 + 1, q_1 + q_2]$ для *

$[M, M - ((p_1 - q_1 + q_2)]$ для /, где M есть 15 в случае *DECIMAL* и 31 в случае *BINARY*

Однако в обозначениях [p, q] разрядность результата p приводится к максимально позволенному значению 15 (в *DECIMAL*) или 31 (в *BINARY*), если p больше этого значения, что может вызвать потерю *старших значащих цифр*.

- когда в выражении участвуют операнды *FLOAT* разрядности p_1 и p_2, разрядность результата равна наибольшему из значений p_1 и p_2 (налицо вопиющая концептуальная ошибка: разрядность в умножении должна быть $p_1 + p_2$).

Применение этих правил порождает некоторые очевидные аномалии:

- *9E0 + 9E0* дает *1E1* (но не *18E0* !)
- *7E0* 7E0* дает *4E1* (но не *49E0* !)
- *10 + 1/2* дает *0.50000* !

Урок состоит в том, чтобы проявлять осторожность с неявными разрядностями констант, и в частности использовать плавающие константы при делении; здесь можно было бы написать:

*9.0E0 + 9.0E0, 7.0E0 * 7.0E0* и *10 + 1/2.0E0*

(практическое правило: в качестве делителя следует всегда использовать *FLOAT*, никогда не следует пользоваться *FIXED*).

д) *Выражения–массивы*

ПЛ/1 отличается от ФОРТРАНа и АЛГОЛа W тем, что позволяет применять операторы (арифметические, операторы отношений и другие) не только к переменным, но и к массивам. В этом случае результат выражения – это массив, полученный применением операторов ко всем элементам массивов, участвующих в выражении.

Рассмотрим объявления

```
DECLARE T(4) CHARACTER (25) VARYING
          INITIAL ('PHYSIQUE', 'STASE',
                   'MATHEMATIQUES', 'LLIQUE');
DECLARE (A(5,4), B(5,4) BINARY FIXED (31,0);
```

Фактически имеются два типа выражений–массивов. Первый тип состоит в применении одной операции с одним заданным операндом ко всем элементам массива.

Например, выражение

'META' | | T

будет иметь значением массив из 4 элементов, получаемый соединением фиксированного *'META'* с каждым элементом из *T:*

'METAPHYSIQUE' 'METASTASE' 'METAMATHEMATIQUES'
'METALLIQUE'

Точно так же *A + A (3, 2)* будет иметь значением массив того же числа измерений и с теми же границами, что и *A,* который получается прибавлением значения *A(3, 2)* ко всем элементам *A*.

Второй тип выражений–массивов получается применением операции к каждой паре соответствующих элементов массивов–операндов. Например,

*A*B*

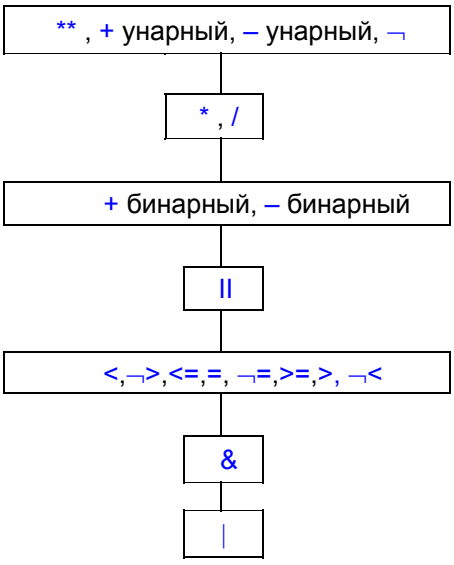
это массив того же числа измерений и с теми же границами, что *A и B,* в котором каждый *(I, J)* – элемент есть произведение соответствующих элементов из *A* и *B*: *A(I, J) * B(I, J)*. Обратите внимание, что речь идет не о «матричном произведении» *A* и *B,* которое здесь не определяется.

Применение выражений–массивов требует некоторых предосторожностей; в частности, все массивы, входящие в такое выражение, должны иметь одну и ту же размерность или одинаковые границы.

В гл. V мы покажем другие методы глобальной обработки массивов в ПЛ/1.

е) *Приоритет операций*

В ПЛ/1 имеет место следующая иерархия операций в порядке убывания приоритетов:



ж) *Встроенные функции*

Некоторые действия могут быть выполнены в ПЛ/1 не с помощью операций, а встроенными функциями, наиболее интересные из которых мы опишем ниже. Мы предположим, что t_1 и t_2 – выражения типа *CHARACTER*; *m* и *n* – арифметические выражения с целыми значениями; A – массив числового типа произвольной размерности.

В приложении Д приведены список встроенных функций ПЛ/1 и выполняемые ими действия. Некоторые из них являются числовыми функциями, например *ABS* (абсолютное значение) или *COS* (косинус), другие – функциями преобразования типов. Вот наиболее оригинальные по отношению к другим языкам функции:

– функции обработки массивов *HBOUND, LBOUND, DIM, SUM* и *PROD:*

- *HBOUND (A,n)* – верхняя граница *n*–го измерения *A*
- *LBOUND (A,n)* – нижняя граница *n*–го измерения *A*
- *DIM(A,n)* – это *HBOUND (A, n)–LBOUND(A, n) + 1*
- *SUM* – сумма элементов *A* (если *A* – числовой массив)
- *PROD* – произведение элементов *A* (если *A* – числовой массив)

Замечание. Функции *HBOUND, LBOUND* и *DIM* на самом деле интересны только для массивов, границы которых могут быть фиксированы при выполнении программы (гл. IV).

– функции обработки строк *SUBSTR* (с результатом строкового типа), *LENGTH, INDEX, VERIFY* (с целым результатом);

- *SUBSTR (t_1, m, n)* имеет своим значением подстроку, содержащую *n* литер из t_1 начиная с *m* –й.

Пример:

SUBSTR('ПРЕДОСТОРОЖНОСТЬ',5,12) дает *'ОСТОРОЖНОСТЬ' SUBSTR* называется псевдофункцией в терминологии ПЛ/1, потому что выражение, включающее *SUBSTR,* может быть использовано как переменная в присваивании, например, чтобы модифицировать часть строки.

LENGTH(t_1) – длина строки t_1. Эта функция особенно интересна для переменных, объявленных *VARYING.*

Пример: *LENGTH ('ПРЕДОСТОРОЖНОСТЬ')* есть *16;*

- *INDEX (t_1, t_2)* служит для определения, является ли t_2 подстрокой t_1. При отрицательном ответе значение функции равно 0. при положительном оно равно самому малому *m,* такому, что

t_2 = *SUBSTR (t_1,m, LENGTH (t_2))*

Пример: *INDEX ('DORABELLA', 'ABEL')* есть *4*
INDEX ('FIORDILIGI,'I') есть *2*
INDEX ('PAPAGENO','PAPAGENA') есть *0*

- *VERIFY (t_1,t_2)* служит для определения, все ли литеры t_1 принадлежат к t_2 (для того чтобы проверить, все ли они буквы, цифры и т.д.). При положительном ответе значение функции равно нулю; при отрицательном ответе – самому малому *m,* при котором *INDEX(SUBSTR(t_1, m, 1), t_2) = 0,* т.е. номеру в t_1 первой литеры, не принадлежащей t_2.

Пример: *VERIEY('TZARA', 'MOZART')* дает *0,*
VERIFY ('31/12/80','0123456789') дает *3.*

II.5.4. Программы, операторы

II.5.4.1. Физическая форма программы

Программа на ПЛ/1 формируется как последовательность 80–литерных строк. Формат свободный; это означает, в частности, что можно разделять основные элементы языка (константы, ключевые слова, идентификаторы и т.д. несколькими пробелами вместо единственного; однако базовые элементы должны описываться без внутренних пробелов (*DECLARE,* но не *DEC LARE*).

Соотнесение текста программы колонкам перфокарт различается в разных операционных системах; обычно используются колонки со 2–й по 72–ю на ИБМ 360 и 370 (исключение составляет колонка 1: в одних системах она служит соглашениям о написании комментариев, в других – условностям языка управления заданиями этих машин).

Комментарий – это последовательность литер, заключенная между литерами /* и */ и не содержащая последовательно расположенных литер * и /:

/ КОММЕНТИРУЕМ, КОММЕНТИРУЕМ */*

Программа ПЛ/1 – это последовательность объявлений и операторов, окаймленная специальными директивами (которые мы не будем пояснять):

PROCEDURE OPTIONS (MAIN); в начале

и

END; в конце

II.5.4.2. Некоторые операторы

а) присваивание

Присваивание записывается как

пер = выр;

где *выр* – выражение, а *пер* – переменная, элемент массива или имя массива (без индекса). В последнем случае *выр* тоже должно быть массивом или выражением–массивом (II.5.3.4.д) и должно иметь то же число измерений и те же границы.

Чтобы выполнить многократное присваивание одного и того же значения нескольким переменным, записывают (не путать с соглашениями АЛГОЛа)

пер1, пер2, ..., перn = выр;

Мы видели, что в ФОРТРАНе и АЛГОЛе W уделено внимание различению знаков операций в операторе присваивания (соответственно = и. :=) и в операторе отношения равенства (*.EQ.* и =). ПЛ/1 смело использует = в качестве того и другого знаков, благодаря чему допустимы операторы

A = B = C;

где (если предположить *A* объявленной *B1T(1),* а *B* и *C* произвольного типа) *A* принимает значение *'1' B* или *'0'B* в зависимости от того, равны ли *B* и *C*. Программа не становится от этого яснее!

Заметим также, что в таком операторе присваивания определение роли знаков = выполняется справа налево, т.е. оператор следует понимать как

A = (B = C)

Напротив, логическое выражение *A = B = C,* тоже совершенно законное, следует понимать (ср. выше II.5.3.4.г) как

(A = B) = C

т.е. определение роли знаков = осуществляется в обратном порядке, слева направо. Напомним, что это логическое выражение дает *'1'B* или *'0'B* в зависимости от того, равняется ли значение *C* после преобразования типов значению *(A = B),* т.е. *'1'B,* если *A* и *B* имеют одинаковое значение после преобразования типов, и *'0'B* в противном случае. Мы оставим читателю заботу выяснить таким же образом смысл присваивания

A=B=C=D

и поразмышлять о достоинствах свободы выбора в понимании языка программирования.

Как и в случае смешанных выражений, при присваивании почти все комбинации типов возможны при соответствующих преобразованиях, и здесь мы также очень мало используем эти возможности.

б) Ввод–вывод

В ПЛ/1 имеются операторы ввода–вывода с форматом, сравнимым с фортрановским, операторы, обрабатывающие файлы и простые операторы чтения и записи, подобные соответствующим операторам АЛГОЛа W. В самом своем элементарном виде они записываются так:

- для чтения:

GET FILE (имяфайла) LIST $(x_1,x_2,...,x_n)$

где *имяфайла* – имя файла и $x_1, x_2, \ldots, x_n$ –имена переменных произвольных типов. В файле с именем *имяфайла* найдутся константы соответствующих типов, значения которых будут присвоены $x_1, x_2 \ldots, x_n$.

Связь между именем *имяфайла* и реальным файлом будет осуществляться посредством языка управления заданиями. В самом простом случае (обычно в случае считывания с карт) можно опустить спецификацию имени файла:

GET LIST $(x_1,x_2,\ldots,x_n)$*;*

- для записи:

PUT FILE(имяфайла)LIST$(e_1,e_2,\ldots,e_m)$*;*

где $e_1, e_2, \ldots, e_m$ – выражения произвольных типов, которые будут записаны в файле с именем *имяфайла* в соответствии с форматом «внешний: (т.е. понятный человеку) и разделены пробелами. В самом простом случае (обычно–вывод на печатающее устройство) можно опустить спецификацию имени файла:

PUT LIST $(e_1,e_2,\ldots,e_m)$ *;*

Библиография

А. ОБЩИЕ РАБОТЫ ПО ЯЗЫКАМ ПРОГРАММИРОВАНИЯ

На появление большого синтезирующего труда по языкам программирования сейчас можно только надеяться. [Саммет 69] – это достаточно полный и подробный каталог языков, известных в США в 1967 г.; [Хигман 67] и [Элсон 73] – общие работы, рассматривающие главные проблемы, связанные с языками программирования. [Николе 75] – это общий и несколько поверхностный обзор, но он особенно ценен обширной прокомментированной библиографией. Недавняя работа [Баррон 77] обсуждает ясно и лаконично некоторые важные аспекты концепции языков.

Б. ФОРТРАН

Существует много учебников начального курса ФОРТРАНа разного качества. На французском языке мы рекомендуем [Мишельанджели 71], который представляет собой введение в язык, и [Куртен 74] – двухтомный начальный курс программирования, использующий «современное» приближение к ФОРТРАНу[1].

На английском [Лармут 73] – углубленное исследование самых оригинальных и наименее известных принципов ФОРТРАНа.

Официальным американским стандартом ФОРТРАНа служит [ANSI 66].

В. АЛГОЛ W

Пересмотренное сообщение документа, определившее АЛГОЛ 60, это [Наур 63]. Введением в АЛГОЛ 60 могут служить на французском [Болье 64] и [Арсак 65], на английском [Дейкстра 61].

АЛГОЛ W был первоначально описан в [Вирт 66]. Официальный документ – [Сайте 72].

[Флойд 70] представляет собой курс введения в программирование, использующий АЛГОЛ W. Более содержательны учебники–[Шион 73] на французском и [Кебурц 75] на английском.

Г. ПЛ/1

Краткое описание ПЛ/1 на французском языке можно найти в [Берте 71]. На английском [Конвей 73] – это «структурированное» введение в программирование, использующее в качестве базового языка ПЛ/С подмножество ПЛ/1, которое устраняет

[1] Среди доступных книг, написанных о ФОРТРАНе на русском языке, упомянем [Первин 72], [Карпов 76], [Салтыков 76], [Ющенко 76]. – *Прим. перев.*

все вычурные свойства языка. Оно дает основу для «разумного» использования ПЛ/1.

Официальным определяющим документом до сих пор остается [IBM 70]. Новый стандарт находится в стадии подготовки[1].

Д. ДРУГИЕ ЯЗЫКИ

Читателю, желающему познакомиться с другими интересными языками программирования, можно рекомендовать следующие книги, которые могут служить введениями в языки[2]:

- АЛГОЛ–60: [Брудно 71]*, [Лавров 72]*;
- ЛИСП: [Сихлоси 76] или [Мак–Карти 62]; на русском – [Лавров 78а]* и [Маурер 76]*.
- АЛГОЛ–68: [АФСЕТ 75] (на французском языке); на русском – [Васильев 72], [Ершов 79], [Пейган 79]*, [Линдси 73];
- БЛИСС: [Вулф 71];
- ПАСКАЛЬ: [Иенсен 75], [Вирт 72], [Вирт 76], [Вирт 71а];
- СИМУЛА–67: [Биртвистл 73]; на русском – [Дал 69]*.
- БЭЙСИК: [Кетков 78]*;
- АПЛ: [Катцан 70]; на русском – [Гилман 79]*;
- КОБОЛ: [КОБОЛ 77]*, [Ющенко 73]*, [Мадженес 79]*.

УПРАЖНЕНИЯ

II.1 Простительная ошибка

В программе, написанной на ФОРТРАНе, обнаружен такой оператор (точно воспроизведенный):

```
      SALF = –.117399E–01+.995239*RW1+.242413E–01*RW1**2–.854861E–03*
     1  RW1**3+.232086E–04*RW1**4–.296266E–06*RW1**5+.171789E–08*RW1**6–
     2  .363143E–11*RW1**7
     3  .363143E–11*RW1**7
```

В последней строке очевидная ошибка – случайное дублирование предыдущей строки, отличающееся литерой продолжения. Однако

- программа была оттранслирована без диагностических сообщений;
- при выполнении программа выдала предусматривающийся результат (во всяком случае, по отношению к значению переменной *SALF*), причем при самых разных значениях.

Почему эта ошибка может считаться простительной?

II.2 Эпсилон

Одна из величин, характеризующих наиболее ощутимым для пользователя образом *машинную арифметику вещественных* (II.1.1.5), носит название *машинного эпсилона*. Это наименьшее положительное число ε, такое, что $1 \oplus \varepsilon > 1$, где $\oplus$ – «модель» сложения, реализуемая устройствами ЭВМ над вещественными числами.

а) Доказать, что действительно существуют ненулевые числа x, представимые в машине значением x, отличным от представления нуля и такие, что наилучшим приближением величины $1 \oplus x$ будет 1 (можно основываться на «мини–системе», приведенной в примере).

[1] На русском языке – [Курочкин 68], [Скотт 77]. – *Прим. перев.*

[2] Звездочкой помечена литература, добавленная при переводе. – *Прим. перев.*

б) Выразить машинный эпсилон как функцию характеристик системы: B (основания) и S (числа значащих битов).

в) Напишите на вашем любимом языке программирования программу, определяющую эпсилон той машины, на которой программа будет выполняться с точностью до множителя 2 (этот вопрос предвосхищает следующую главу).

ГЛАВА III.
УПРАВЛЯЮЩИЕ СТРУКТУРЫ

Меня можно упрекнуть в несколько извращенной тяге к системе:
Даниил Столпник неплохо жил на своем столбе, он даже ухитрился превратить его (очень нелегкое дело!) в структуру.

Роллан Барт

Тутти Системати
во Фрагментах из Любовных Речей

УПРАВЛЯЮЩИЕ СТРУКТУРЫ

Описав базовые элементы языков программирования, и в частности основные операторы, мы теперь должны изучить способы комбинирования таких операторов для построения программы. Это позволяют сделать управляющие структуры. После описания широких классов управляющих структур и их представлений в рассматриваемых языках мы покажем методологическую необходимость придерживаться достаточно строгой дисциплины в выборе управляющих структур.

III.1. Введение

Целью этой главы является попытка классифицировать и характеризовать методы, которые позволяют программисту полностью контролировать действия, предписываемые его программами.

Преимущество вычислительных машин – но в то же время и трудность их использования – состоит в том, что они без труда выполняют операции, которые весьма трудоемки или даже практически невыполнимы людьми. И наоборот, действия, которые нам кажутся столь очевидными, что мы не пытаемся их анализировать, зачастую становятся проблемой, когда предпринимается попытка их запрограммировать. Априорно заманчивая возможность, предписав одним росчерком пера выполнение тысяч элементарных действий, ставит трудную проблему *понимания сложности,* проблему, для решения которой необходимо вооружить себя определенными средствами. Мы рассмотрим здесь эти средства, которые носят название управляющих структур.

Как всякое автоматическое устройство, вычислительная машина способна выполнять некоторое число элементарных операций: сложение, вычитание, присваивание, сравнение и т.д. В отличие от обычной машины она обладает способностью самостоятельно управлять выполнением этих операций во времени.

Именно этот аспект будет интересовать нас в данной главе: как можно указать с помощью программы распорядок работы этих операторов? Может ли **статический объект** – текст программы – быть построенным таким образом, чтобы дать ясное представление о **динамической ситуации** – ее выполнении?

Если попытаться явно выразить возможности управления вычислительной машиной, можно выделить три основные категории управляющих структур, которые мы будем называть *цикл, ветвление* и *цепочка.* Эти комбинируемые по желанию базовые схемы дают нам «механический конструктор», позволяющий описать процесс вычислений произвольной сложности, сохраняя отчетливое видение структуры процесса и каждой из его компонент.

III.2. Обозначения

Прежде чем описывать собственно управляющие структуры, необходимо ввести несколько фундаментальных понятий, связанных с анализом программ.

III.2.1. Состояние программы

Функционально программа определяется как отношение между ее возможными *данными* и соответствующими *результатами*[1]. Практически эта программа размещена в памяти ЭВМ; в некоторый момент ее выполнения она занимает область этой памяти; один из операторов программы активен, и каждый из ее элементов обладает некоторым значением, быть может еще не определенным.

Состояние программы в некоторый заданный момент ее выполнения характеризуется, таким образом, двумя типами сведений (Рис. III.1):

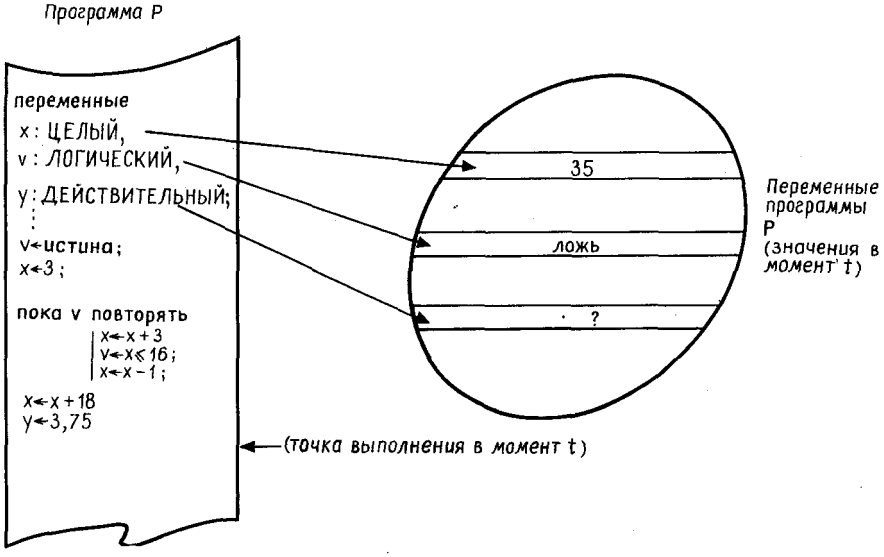


Рис. III.1 Состояние программы.

- информация, определяющая значения величин во время выполнения программы (физически это состояние некоторой области памяти), классифицируемых как «данные», «промежуточные переменные», «результаты»; мы будем говорить, что это переменные программ;

- информация, указывающая «активную точку» (или активные точки) программы в некоторый момент; на практике часто бывает удобно (подпрограммы, составные операторы) или необходимо (параллельные процессы) полагать, что имеют дело с несколькими активными точками; мы ограничимся, Однако, для простоты

[1] Математически речь идет о *частичной* функции, т.е. результат может быть неопределенным для некоторых данных (которые порождают бесконечное зацикливание программы).

алгоритмическими процессами, протекание которых можно описать с помощью единственной точки выполнения, отмечающей в каждый момент времени выполнение активного оператора.

Эти замечания позволяют прояснить природу двух основных классов действий, которые может выполнять вычислительная машина :

- Действия «первого типа» выделяют область памяти (переменные) программы, но не имеют дела с точкой выполнения.

  Так, взятое изолированно присваивание ФОРТРАНа

  I = 5

  которое дает значение 5 целой переменной *I*, принадлежит к этой категории, так же как и все рассмотренные в предыдущей главе операторы (присваивание, ввод, вывод).
- Действия «второго типа» не влияют на переменные программы, но меняют ее точку выполнения. Так, в ФОРТРАНе к ним относятся

  IF (I.EQ.5) GOTO 100

  или *GOTO 200*

Действие такого типа указывается также и неявно, когда, например опять в ФОРТРАНе, один оператор следует за другим: если только это не ветвление, можно рассматривать, что точка выполнения меняется так, что задает следующий оператор. *Замечание:* Оператор ФОРТРАНа

Z=F(X,Y)

где *F* – функция (ср. гл. IV), прерывает последовательность команд на машинном уровне. На уровне фортрановской программы это прерывание не проявляется; возможно, меняются только *X, Y, Z* (и, может быть, другие переменные). Мы будем считать поэтому, что речь идет о действии первого типа.

На практике многие операторы языка могут менять сразу и точку выполнения, и переменные (например, в ФОРТРАНе *IF(F(X,Y).GT.100.)GOTO100)*. Тем не менее для анализа удобно разделять всякий сложный оператор на действие первого типа (модификация переменных) и следующее за ним другое действие второго типа (модификация точки выполнения). Так,

IF(F(X,Y).GT.100.) GOTO 100

эквивалентно

Z = F(X, Y)

за которым следует *IF (Z.GT.100.) GOTO 100*,

где *Z* – переменная, которая фактически не появляется.

III.2.2. Блок–схемы

Действия первого типа (изменения переменных), которые мы рассмотрели в предыдущей главе, относительно просты и хорошо понятны. Действительно, мы полагаем, что все они (см. гл.II, и в частности II.2):

- либо операторы ввода и вывода вида

  читать(ф) $x_1x_2,...,x_n$

  или

  писать(ф) $y_1,y_2,\ldots, y_n$
- либо присваивания вида

$Z \leftarrow f(Y)$

(Т. е. «дать переменной Z значение f(Y)»)

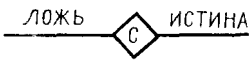
где Z и Y – переменные (или массивы, или объекты сложных типов, которые мы увидим в гл. V) типа «результат» или «промежуточная переменная» для Z либо типа «данное» или «промемежуточная переменная» для Y, а f – функция, разложимая на известные операторы[1]. Необходимость введения управляющих структур следующего раздела вытекает из того, что предписания второго типа (которые влияют на порядок выполнения других операторов программы) являются слишком общими и, значит, трудно усваиваемыми, если разрешены свободные манипуляции с точкой выполнения.

Мы проиллюстрируем эти структуры с помощью условных рисунков, называемых блок–схемами. Эти схемы строятся с помощью четырех следующих элементов:

а) стрелки

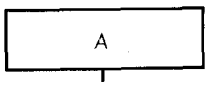
для изображения безусловного перехода точки выполнения;

б) «блоки» с двумя выходами



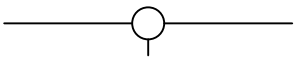
изображающие условный переход точки выполнения, т.е. переход по одной или другой стрелке в зависимости от условия C, которое является функцией переменных программы и может иметь значение **истина** или **ложь**;

в) прямоугольные «блоки» с одним выходом

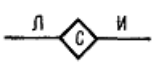


изображают действия A первого типа (изменения переменных);

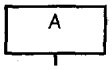
г) узлы с двумя входами и одним выходом:



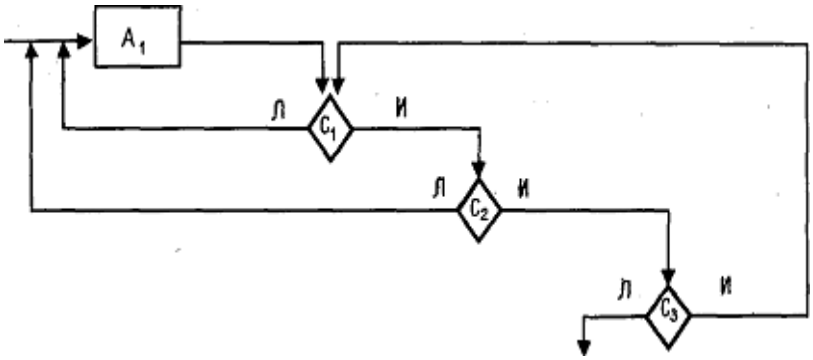
предназначенные для двух путей перехода точек выполнения; из такого узла может выходить только одна стрелка. Узлы позволяют требовать, чтобы блоки типа



или

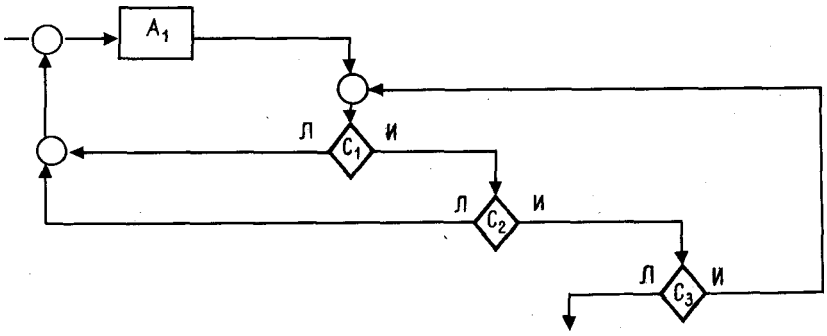


имели на входе только по одной стрелке. Поэтому мы не будем рисовать



считая правильным изображение

[1] Разумеется, в связи с действиями первого типа неизбежно возникают проблемы : спецификация порядка чтения, природа составляющих f операторов, соответствие типов, природа «известных операторов» и т.д. Мы будем считать эти вопросы вторичными по отношению к тому, чем мы занимаемся здесь.



Эти четыре элемента – стрелки, блоки с двумя выходами, прямоугольные блоки и узлы – будут использоваться для пояснения вводимых управляющих структур[1]. Важно заметить, что мы не хотим защищать использование блок–схем более общего типа. Блок–схемы были и остаются вспомогательным средством документирования программ. Мы полагаем, что вводимые ниже структуры, использованные соответствующим образом, позволят избавиться от необходимости такого типа документации: они на самом деле имеют такую же внутреннюю выразительность, что и блок–схемы, и позволяют программисту отчетливо видеть структуру алгоритма в самом тексте программы. Проблема документации будет рассмотрена в гл. VIII (VIII.4.3.4).

III.3. Базовые структуры

III.3.1. Циклы

Как известно, вычислительные машины способны (в противоположность людям) многократно без видимой усталости повторять одинаковые или похожие действия.

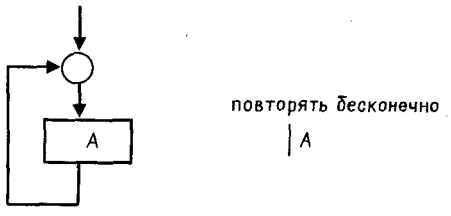
Поэтому естественно ввести в качестве первого типа управляющей структуры повторение в его различных формах (по этому поводу мы могли бы в равной степени употреблять слова «повторения» или «цикл»).

III.3.1.1. Бесконечный цикл

Тип цикла, который сразу же приходит на ум, это *бесконечный цикл;* мы будем его обозначать

повторять бесконечно
| действие

и иллюстрировать блок–схемой



Рассмотрим, например, вычислительную машину, управляющую электрической цепью и способную выполнять операцию:

проверить напряжение сети

Напряжение в принципе должно проверяться постоянно; следовательно, речь идет о бесконечном цикле («бесконечный» означает на практике «до ближайшей

[1] Для полноты следовало бы ввести специальные «блоки» типа «начало выполнения» и «конец выполнения». Но для описания базовых структур, которые представляют собой предназначенные для последующих объединений элементы программ с одним входом и не более чем одним выходом, нам они будут не нужны.

аварии»). Тогда программа машины записывается

повторять бесконечно
| проверить напряжение сети

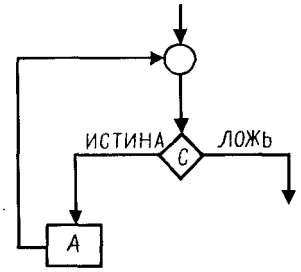
Нам придется рассматривать главным образом конечные процессы, и мы будем только изредка упоминать бесконечные циклы. Тем не менее не следует пренебрегать этой управляющей структурой, которая играет фундаментальную роль в области процессов реального времени, операционных систем и т.д. Пример области, в которой такой цикл является удобным понятием, это «сопрограммы» (IV.7).

III.3.1.2. Циклы, управляемые условиями (типа пока и до)

В обычных условиях, однако, бывает необходимо повторять некоторое действие не бесконечно, а только *пока* сохраняет силу некоторое условие. Такой оператор будем обозначать

пока условие **повторять**
| действие

Условие – это *логическое выражение* (т.е. выражение, способное принимать одно из двух значений **истина** и **ложь**), зависящее от переменных программы. Ясно, что выполнение такого цикла может закончиться, только если действие способно изменить один из элементов, входящих в условие, или если оно было ложным с самого начала (в последнем случае цикл не выполняется ни разу).



пока c повторять
| A

Заметим, что цикл «пока ... повторять» без труда моделирует цикл «повторять бесконечно»:

пока истина повторять
| действие

Цикл типа пока может также использоваться, когда необходимо достичь состояния программы, при котором некоторое условие c, зависящее от переменных программы, становится истинным. Если известно некоторое действие a, которое интуитивно «приближает» начальное состояние к состоянию, где c истинно, пользуются оператором:

пока ~c **повторять**
| a

(~c, читаемое «не c», объясняет условие, обратное c, т.е. **истина,** если c **ложь** и наоборот).

Видно, что главная проблема, которая возникает в связи с циклом типа **пока,** это проблема его **окончания:** как можно гарантировать, что цикл завершится после некоторого числа итераций, для некоторого класса данных? Эта проблема не имеет решения в общем случае; на практике, однако, часто бывает возможно найти свойства a и c, обеспечивающие окончание. В разд. III.4 мы обсудим эту проблему подробнее.

В качестве простого примера цикла **«пока ... повторять»** (у нас будут поводы встретиться и с многими другими примерами в этой книге) представим вычислительную машину, снабженную магнитной лентой. Лента содержит данные о студентах некоторого вуза; машине поручено вычислить размер стипендии для каждого из студентов. Наша машина умеет определить, имеет ли она что–нибудь для прочтения на ленте в той точке, где она находится (т.е. не добралась ли она до конца ленты), и в этом случае выполнить оператор

вычислить стипендию и перейти к следующему студенту

Тогда ее программа запишется

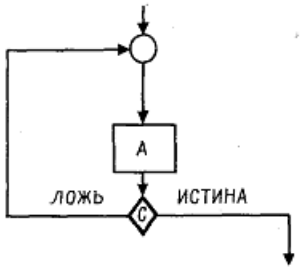
пока есть что–нибудь на ленте **повторять**
| назначить стипендию и перейти к следующему студенту

Заметим, что если лента исходно пуста, то не предпринимается никакого действия. Во всяком случае, после выполнения цикла оставшаяся впереди читающей головки часть ленты будет пуста.

Другая форма повторения, близкая к предыдущей, это цикл

повторять
| действие
до условие

который, выполнив действие один раз, повторяет его до тех пор, пока условие станет верным:



повторять
|A
пока c

Здесь в отличие от предыдущего случая действие всегда выполняется, по крайней мере один раз, каким бы ни было начальное значение условия, т.е. действие

повторять
| A
до C

эквивалентно действию A, за которым следует

пока ~ C **повторять**
| A

III.3.1.3. Цикл с параметром

Полезным типом цикла является так называемый цикл *с параметром,* который мы будем обозначать

для всякого элемента x **принадлежащего** M **выполнить**
| действие

где x – это «параметр» цикла (обычно используемый в действии), а M – множество. x играет роль связанной переменной в математике.

Мы будем использовать иногда понятие, являющееся смесью цикла пока и цикла с параметром:

для x **принадлежащего** M **пока** условие **повторять**
| действие

Эта управляющая структура предполагает, что порядок выборки элементов из множества M известен. Приведенный выше оператор рассматривает один за другим элементы из M; если их больше нет, или если элемент, рассматриваемый в настоящий момент, не удовлетворяет условию, то цикл заканчивается; в противном случае выполняется действие и осуществляется переход к следующему элементу.

Большинство старых языков программирования предполагает в качестве цикла типа «с параметром» только **цикл со счетчиком**, где M – конечное множество с естественным порядком, определяемым подмножеством целых чисел. Этот тип цикла обозначают:

для x **от** m **до** n **повторять**
| действие

где m и n – целые. Можно также задать *шаг* изменения x (неявно принимаемый равным единице в только что приведенном цикле):

для x **от** m **до** n **шаг** p **повторять**
| действие

Мы увидим, что в зависимости от языка цикл со счетчиком понимается как цикл типа

пока ... **повторять** ...

или **повторять** ... **до** ...

т.е. действие либо выполняется обязательно, по крайней мере один раз, либо может и не выполняться. Со своей стороны мы покажем, что этот тип цикла эквивалентен нулевому действию, если m > n и p > 0 или если m < n и p < 0 (ср. обсуждение в III.5.2.3).

III.3.2. Ветвления

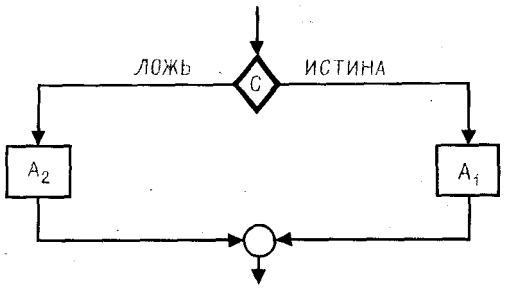
Другая причина популярности (или непопулярности) вычислительных машин состоит в их способности делать *выбор* между несколькими решениями в зависимости от значения некоторого параметра.

III.3.2.1. Альтернатива

Простейшая форма ветвления – это *альтернатива,* где есть два возможных пути и выбор зависит от того, верно или неверно некоторое *условие.* Альтернатива обозначается так:

если условие **то**
| действие 1
иначе
| действие 2

что соответствует блок–схеме

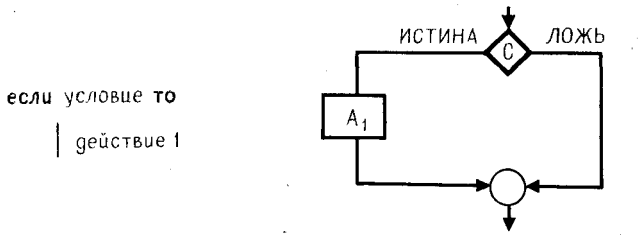


если C то
| A1
иначе
| A2

Теперь, если вернуться к приводимому выше примеру контролера электрической сети и предположить, что он умеет отправлять сообщение дежурному электрику, то его оператор, проверяющий напряжение сети, запишется

если напряжение сети > предельное напряжение **то**
| отправить дежурному сообщение **"ВНИМАНИЕ, НЕ ЗЕВАЙ**!"
иначе
| ничего не делать

В этом примере действие, выполняемое, когда условие ложно (т.е. действие, следующее за иначе), есть пустое действие. Тогда следует опустить иначе и то, что за ним следует, записывая просто:



Тогда программа нашего «контролера» становится такой:

если напряжение сети > предельное напряжение **то**
| отправить дежурному сообщение "ВНИМАНИЕ, НЕ ЗЕВАЙ!"

III.3.2.2. Многозначное ветвление

Часто приходится выбирать не из двух, а из нескольких возможностей. Такую ситуацию называют *многозначным ветвлением* (или *переключателем*) и записывают

выбрать
| условие 1: действие 1
| условие 2: действие 2
| …
| условие n: действие n
иначе[1]
| действие 0

(1)

Этот порядок предусматривает, что выполняется действие, такое, что соответствующее условие i **верно**; или, если ни одно из условий i (i = 1, 2, ..., n) неверно, выполняется действие 0.

Предположим сначала, что «условия» выбраны таким образом, что никакие два из них не могут быть верными одновременно (часто имеет место случай, когда для i = 1, 2, ..., n условие i есть условие «x = i», где x – некоторое выражение, имеющее целое значение). В этом случае формула (1) будет всегда давать такой же результат, что и

если условие 1 **то**
| действие 1
иначе если условие 2 **то**
| действие 2
иначе если ...(2)
...
иначе если условие n **то**
| действие n
иначе действие 0

На машинном уровне, однако, выбор по одному из несовместимых условий (1) может быть реализован более эффективно, нежели последовательность проверок (2) (см. III.5.3.2: «таблица переходов»).

Если несколько условий могут быть верными одновременно, то (1) и (2) перестают быть эквивалентными: в самом деле, в (2) условия проверяются в порядке 1, 2, ..., n; первое верное условие запускает соответствующее «действие», в то время как мы в (1) старались воздержаться от определения проверок условий; этот порядок зависит от соглашений языка и его реализации на конкретной машине. В некоторых задачах интересно было бы, чтобы язык допускал *недетерминированное* выполнение; при этом выбор выполняемого действия (среди тех, соответствующие условия которых верны) случаен, или, точнее, неконтролируем программистом.

---

[1] В оригинале **иначе** в конструкции **выбрать** и **иначе** в конструкции **если** орфографически отличаются (**autrement** и **sinon** соответственно). Однако эти разные в синтаксическом отношении **иначе** без труда распознаются по контексту как транслятором, так и человеком (особенно при структурировании текста программы с помощью отступов). Поэтому, уступая сложившейся в отечественной литературе терминологии, в переводе иначе используется в обеих конструкциях в равной мере. *–Прим. перев.*

В качестве примера, если бы мы хотели развернуть алгоритм «назначения стипендии», по которому работает наша «стипендиальная комиссия»[1], нам, конечно, нужна конструкция типа **выбрать** ... **иначе**. Тогда мы могли бы прийти к чему–то вроде:

```
выбрать
    средний балл = 5: выбрать
                        участие в научной работе: назначить именную стипендию А,
                        участие в общественно–политической работе: назначить именную
                        стипендию Б,
                        участие в спортивно–массовой работе: назначить именную
                        стипендию В
                    иначе назначить повышенную стипендию
    4 ≤ средний бал ≤ 5:
                    если нет троек то
                        назначить стипендию иначе
                        выбрать
                            доход на члена семьи < 60 руб.:
                                назначить стипендию
                            активный общественник:
                                назначить стипендию
                            имеет детей: обработка по специальной программе
                    иначе отказать
    3.5 ≤ средний балл ≤ 4:
                        выбрать
                            доход на члена семьи < 60: назначить стипендию
                            имеет детей: обработка по специальной программе
                    иначе отказать
    средний балл < 3.5: отказать
иначе ошибка в определении среднего балла
```

Вариант конструкции выбора предложил Э. Дейкстра (см. [Дейкстра 76]). Мы будем его обозначать

```
выбрать и повторять
    условие 1: действие 1,
    условие 2: действие 2,
    ...
    условие n: действие n
```

Речь идет о многозначном переключателе, скомбинированном с циклом. Действует он так: если ни одно из условий i не верно, ничего не происходит; иначе выполняется одной из действий i, для которых условие i верно, и все начинается сначала. Эта конструкция легко моделируется предыдущими примитивами: достаточно использовать «**выбрать** ...» предыдущего типа в теле цикла «**пока**». Она весьма типична в некоторых программах, где необходимы повторения до тех пор, пока не выполнено некоторое число условий. Правда, в наше время ни один из «производственных» языков программирования не предлагает реализации «цикла Дейкстры» в том виде, в котором он был сформулирован.

III.3.3. Цепочка

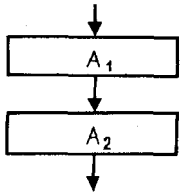
Цепочка – настолько естественная управляющая структура, что иногда забывают

[1] Приведенный здесь пример алгоритма, который может моделировать работу воображаемой студенческой стипендиальной комиссии, заменяет (с согласия авторов) пример из оригинала, где вычисляется величина налога по весьма специфичной и потому мало информативной для советского читателя французской налоговой системе. – *Прим. перев.*

ее фундаментальный характер. Идея соединения действий в цепочку состоит в том, что если предписания действие 1 и действие 2 задают выполнимые действия, то обозначение

действие 1; действие 2

указывает, что действие 2 выполняется вслед за действием 1:



Обозначение, использующее точку с запятой для изображения объединения в цепочку по аналогии с ее функцией в обычном языке, было введено в АЛГОЛе 60. Разумеется, она позволяет выразить объединение более чем двух «действий»:

действие 1;
действие 2;
…
действие n – 1;
действие n

Главный интерес понятия цепочки в том, что результирующий *составной оператор* играет ту же роль, что и простой оператор. Таким образом, можно писать, например,

пока условие **повторять**
| действие 1;
| действие 2;
| действие 3

где вертикальная черта использована для того, чтобы указать на это единство *цепочки* действие 1; действие 2; действие 3.

В качестве примера применения цепочки еще раз вернемся к нашему «контроллеру электросети». Он будет, несомненно, не единственным пользователем линии связи с дежурным электриком; «отправить сообщение» потребует поэтому сначала подключения к линии, и более подробная версия программы запишется:

повторять бесконечно
если (напряжение сети < предельное напряжение) **то**
| подключиться к линии;
| отправить сообщение "ВНИМАНИЕ, НЕ ЗЕВАЙ!";
| освободить линию

III.3.4. Подпрограммы

Часто случается, что программа требует многократного выполнения одного и того же действия с теми же самыми переменными или с различными переменными. Такое действие может быть также общим для ряда программ. Например, в обычном вычислительном центре многие программы должны ежедневно сортировать данные: тип этих данных, их число, «ключи», по которым они должны быть упорядочены, могут существенно различаться; основная задача остается, однако, той же, и можно построить зависящий от параметров алгоритм, который будет способен обрабатывать большую часть таких случаев.

В этих условиях было бы абсурдным, если бы каждая сортировка требовала написания собственной программы. Вместо этого вызывают (одним–единственным оператором) **подпрограмму** сортировки, написанную раз и навсегда, которую снабжают в качестве *аргументов* (или «параметров») сортируемыми данными и их

характеристиками и которая будет после выполнения выдавать обратно эти данные, упорядоченные должным образом.

Подпрограмма является, таким образом, *компонентом программы,* обладающим именем, которое позволяет этой программе (или всякой другой программе) *вызвать* его, чтобы заставить выполнить некоторое *действие* или вычислить некоторые *значения.*

Это определение позволяет нам представить понятие подпрограммы уже в другом свете. В действительности вовсе не обязательно, чтобы задача относилась к типу «часто повторяющихся действий или действий, общих для многих процессов», для того, чтобы она оправдывала написание подпрограммы; достаточно, чтобы она образовывала логически однородную сущность. Рассматриваемая с такой точки зрения подпрограмма есть удобное средство абстрагирования; так, если даже наша машина пока что не реализует оператор «назначить стипендию и перейти к следующему студенту», мы можем допустить, что подпрограмма

пока остается что–то на ленте **повторять**
| назначить стипендию и перейти к следующему студенту

корректна, считая, что вторая строка и есть вызов подпрограммы, которая будет выполнять желаемые действия. Это позволяет нам при написании программы отложить на более поздний срок технические подробности реализации этой задачи.

> Подпрограмма, следовательно, играет в какой–то степени ту же роль, что в математике «теорема» или «лемма»: когда нужно доказать результат в некоторой специальной области, например в дифференциальной геометрии, вовсе не обращаются каждый раз к базовым теоремам теории множеств, а опираются на теоремы, которые косвенно через несколько уровней абстракции основываются на этих аксиомах. Точно так же в информатике прибегают к написанию программы из более простых модулей, в свою очередь детализируемых до тех пор, пока не будет достигнут уровень, на котором команды воспринимаются машиной либо непосредственно, либо с помощью программы «компилятора» или «интерпретатора». Мы вернемся в гл. VIII к этой *нисходящей* концепции программирования и к проблемам, которые она ставит.

Понятие подпрограммы, практически столь же старое, как понятие вычислительной машины, порождает некоторое число проблем, таких, как способ передачи параметров, возможность рекурсивного вызова и т.д., которые будут подробно изучены в гл. IV и VI. С точки зрения структуры базовых алгоритмов, которая нас интересует здесь, подпрограмма – это просто «модуль» (термин, определяемый со всей строгостью в гл. VIII), который, например, может задавать действие или условие во фрагменте программы:

пока условие **повторять**
| действие

Теоретически понятие подпрограммы ничего не добавляет к множеству предыдущих алгоритмических структур, потому что выполнение подпрограммы эквивалентно выполнению вызывающей программы, в текст которой включен текст подпрограммы с подходящей заменой параметров[1]. Однако это понятие играет фундаментальную роль в силу непосредственно экономических соображений (сокращение размеров программ) так же и потому, что оно облегчает составление и восприятие программ.

---

[1] Более тонким является случай рекурсивных программ (см. разд. VI.3).

III.3.5. Композиции базовых структур

Мы уже видели неявное использование понятия композиции примитивных структур; например, каждое действие в

повторять действие **до** условие (цикл)

или в

| действие 1;
| действие 2;
| действие 3 (цепочка)

может быть либо «простым» действием (типа «чтение», «запись» или «присваивание»), либо сложным действием, получаемым с помощью примитивов, как, например, цикл или цепочка действий, которые могут быть в свою очередь сложными, и т.д.

Композиция указанных примитивных структур достаточна для описания любого алгоритмического процесса. Можно утверждать больше: **цепочка** *и* **цикл** *типа* **пока** ... **повторять** ... *действительно принципиально достаточны, чтобы описать действие программы с произвольной блок–схемой* (построенной из элементов, описанных в разд. III.2.2).

Не пытаясь доказывать этот общий результат, посмотрим, например, как можно было бы обойтись без альтернативы **если** ... **то** ... **иначе**.

Пусть лог1 и лог2 – логические переменные (имеющие два возможных значения **истина** и **ложь**). Конструкция

```
лог1 ← условие; лог2 ← ~условие;
пока лог1 повторять
        | действие 1;
        | лог1 ← ложь;
пока лог2 повторять
        | действие 2;
        | лог2 ← ложь
```

эквивалентна (т.е., будучи примененной к тем же данным, дает тот же результат) конструкции

```
если условие то
   | действие 1
иначе
   | действие 2
```

Действительно, ясно, что в (3) только первый цикл будет выполнен, если условие имеет значение **истина**, и только второй, если условие **ложно**; при этом оба они будут выполнены только по одному разу, так как управляющая переменная цикла пока сразу же принимает значение **ложь**. Таким образом, конструкция (3) вполне моделирует альтернативу.

Разумеется, альтернатива **если** ... **то** ... **иначе** – обозначение более выразительное, чем (3), но может быть полезно знать, что речь идет о конструкции, теоретически излишней.

III.3.6. Управление с помощью таблиц

К множеству рассмотренных до сих пор методов управления выполнением программ следует добавить одну особую технику – управление с помощью таблиц, которая не вносит ничего нового с теоретической точки зрения, но часто бывает полезной. Идея этого метода состоит в том, чтобы доверить часть управления не прямо программе, а **данным**. Мы проиллюстрируем его достаточно выразительным примером некоторых задач, выполняемых трансляторами (см. также упражнение III.6).

Пусть необходимо реализовать одну из функций «лексического анализатора» на некотором языке, сходном с ПЛ/1, АЛГОЛом W или ФОРТРАНОМ, а именно

построить программу, способную выделить в тексте анализируемой программы следующие элементы, или лексемы:

- целые константы, положительные или нулевые, – примеры: *423 0 8492*
- вещественные константы – *примеры*: *24.37 0.64*
- знаки операций – *примеры*: = ( ) + –
- идентификаторы – *примеры*: *X Y1 T02A PAPA* (длина идентификаторов не ограничивается)
- разделители: серии из одного или нескольких пробелов или «переводов каретки» (окончание строк)

Каждое выполнение нашей программы должно выделять из текста очередной такой элемент и выдавать два данных: код, указывающий его тип (константа, целая или вещественная, идентификатор и т.д.), и строку, образованную составляющими его литерами (*"423", "T02A"* и т.д.). Фактически будет написана программа для выполнения такой задачи, но мы еще не показали соответствующие обозначения).

Предположим, что мы располагаем «операцией»

читать–лит

действие которой состоит в прочтении очередной литеры анализируемого текста и в выдаче ее значения; так, если с имеет тип ЛИТЕРА, то присваивание

с ← читать–лит

прочитает новую литеру и присвоит ее значение с.

Со всякой литерой с связывают код, обозначаемый

класс (с)

который принимает одно из пяти значений:

«цифра»	(для цифр *0, 1, 2, …, 9*)
«точка»	(для литеры «.»)
знак операции	(для +, –, : и т.д.)
«пробел»	(для пробела и перевода каретки, которые обрабатываются операцией читать–лит как обычные литеры)
«буква»	(для *A, B, …, Z* и всех прочих литер).

Эти значения (которые на практике могут изображаться числовыми кодами) характеризуют класс, к которому принадлежит анализируемая литера.

Наконец, нужно, чтобы в начале каждого выполнения программы нашего «лексического анализатора» переменная след–литера типа ЛИТЕРА имела своим значением первую литеру, не принадлежащую уже проанализированному элементу; тот же смысл переменная след–литера должна сохранять и после каждого выполнения программы. Это предполагает, очевидно, соответствующую инициализацию.

Реализуемая программа может быть изображена **диаграммой переходов** (Рис. III.2). Диаграмма используется следующим образом. Исходным является состояние 1 (состояния пронумерованы числами в квадратиках). На каждом этапе читается новая литера, и в зависимости от ее «класса» надо двигаться по соответствующей стрелке. Так, отправившись из состояния 1 и прочитав литеру 9, которая является цифрой, направляются по стрелке, помеченной «цифра» и приводящей в состояние 2. Если стрелка не приводит ни к какому состоянию (как стрелка, помеченная «все, кроме цифры» и выходящая из состояния 3), алгоритм заканчивается: проанализирован элемент, тип которого определен последним встретившимся состоянием (этот тип указан на диаграмме под номером соответствующего состояния).

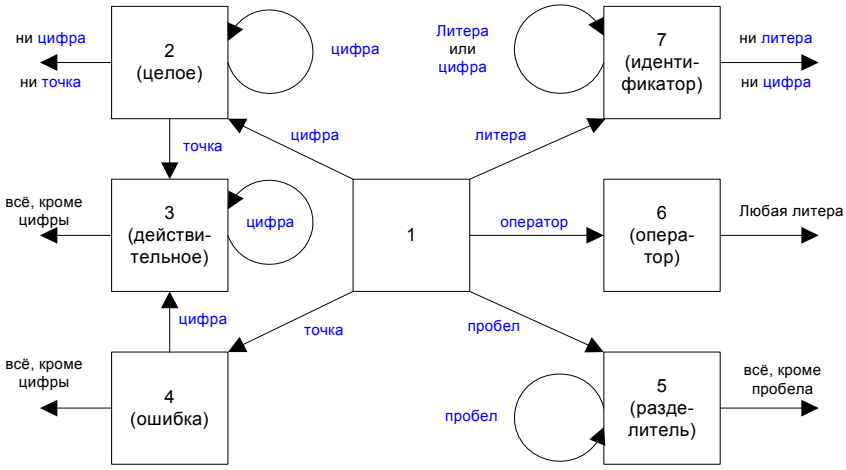


Рис. III.2 Диаграмма перехода.

Читателю предлагается проверить на нескольких примерах правильность диаграммы и заодно уточнить определения распознаваемых элементов: может ли быть опущена часть числа, следующая за десятичной точкой в вещественном числе, и т.д.? Диаграмму можно представить массивом:

массив Переход [1 : 7, 1 : 5] : ЦЕЛЫЙ

значениями которого служат данные Рис. III.3 (7 – число состояний диаграммы; 5 – число «классов литер»).

Переход [n, cc] = m означает просто, что на диаграмме Рис. III.2 стрелка, выходящая из состояния n с меткой cc (имя класса), приводит в состояние m или не приводит ни к какому состоянию, если m = 0 (например, случай, когда из диаграммы «выходят» в состоянии n = 5 и когда встречается что–либо, кроме пробела).

Когда массив Переход задан, анализ выполняется с помощью приводимой ниже программы (напомним, что обозначение $t_1|\ |t_2$ представляет конкатенацию двух строк или литер t_1 и t_2, получаемую их непосредственным присоединением друг к другу; так, конкатенация *"TAKE"* и *"YAMA"* дает *"TAKEYAMA"*):

```
переменные след–литера: ЛИТЕРА,
          | имя: СТРОКА {содержит литеры, которые составляют
          | распознаваемый элемент }
          | состояние, след–состояние: ЦЕЛЫЕ;
имя ←""; { инициализация пустой строкой }
состояние ←1;
{предполагают, что след–литера имеет значением первую литеру, не принадлежащую к
уже проанализированному элементу}
след–состояние ← переход [состояние, класс (след–литера)],
повторять
          | имя ← имя| | след–литера;
          | состояние ← след–состояние;
          | след–литера ← читать–лит {чтение новой литеры};
          | след–состояние ← переход [состояние, класс (след–литера)] до след–
          | состояние = 0
{при след–состояние = 0 элемент опознан: имя элемента определено переменной имя, а
его тип указан целым состояние с учетом соответствия, задаваемого Рис. III.2. след–
литера снова имеет значением первую литеру, не принадлежащую анализируемому
элементу; это свойство инвариантно}
```

Класс новой литеры / Предыдущее состояние	«цифра»	«точка»	«Знак операции»	«буква»	«пробел»
1	2	4	6	7	5
2	2	3	0	0	0
3	3	0	0	0	0
4	3	0	0	0	0
5	0	0	0	0	5
6	0	0	0	0	0
7	7	0	0	7	0

Рис. III.3. Таблица переходов.

Этот пример позволяет сформулировать принципы управления с помощью таблиц: статическая структура программы здесь предельно проста, она дает мало указаний на динамику ее выполнения; в самом деле, программа довольствуется интерпретацией управляющей таблицей. Здесь это массив Переход, который и является настоящим ключом к пониманию алгоритма (очевидно, это простой перевод диаграммы Рис. III.3).

Преимущества и недостатки этого метода ясны. В его активе следует отметить гибкость результирующей программы; так, если необходимо в вышеприведенном примере разрешить пробелы внутри идентификаторов, достаточно заменить семеркой значение 0 в элементе [7, «пробел»] массива Переход; нет необходимости в модификации программы, которая может с таким же успехом использоваться с самыми различными таблицами (см. упражнение III.8). Именно в силу этой гибкости метод управления с помощью таблиц, называемых также «таблицами решений», часто используется в таких областях, как компиляция или программное обеспечение автоматизированных систем управления.

С другой стороны, однако, программа существенно теряет в ясности в том смысле, что становится очень трудно представить динамику ее поведения. В нашем случае таблица содержит 35 элементов, из которых только 11 ненулевые, так что относительно просто ответить, скажем, на такой вопрос, как: «распознает ли эта программа знаки операций, состоящие более чем из одной литеры?» В некоторых таблицах решений, применяемых в обеспечении АСУ, приходится выбирать из сотен или тысяч возможностей. Обыкновенному человеку в принципе невозможно контролировать такие процессы. Единственное средство их анализировать и «структурировать» так, чтобы они оставались воспринимаемыми, это разложить их на строго выбранные примитивы – управляющие структуры.

III.4. Свойства базовых структур: введение в формальную обработку программ

III.4.1. Введение и обозначения

Базовые структуры были ранее введены более или менее интуитивно: мы объясняли их действие то «блок–схемами», то просто на словах. Настоящая секция посвящена более систематическому изучению свойств этих структур. Действительно, Хоар, который первым исследовал проблему, показал, что свойства, *выводимые* из интуитивного определения управляющих структур, могут вместо этого быть основанными на *строгом определении*. Они, следовательно, играют очень важную

теоретическую роль.

Практический же интерес этих свойств в том, что они позволяют программисту рассматривать свою программу как совершенно определенный и допускающий обработку объект и, исходя из *текста* программы, **доказывать** некоторые свойства ее *выполнения.* Это решающий шаг в решении фундаментальной проблемы, которую мы уже имели повод упомянуть: как установить отношение между *статическим* аспектом и *динамическим* поведением программы. Помощь, которую нам оказывают введенные выше управляющие структуры, – весьма весомый аргумент в пользу их преимущественного использования.

Все приводимые ниже результаты имеют общую форму:

> *если* свойство P изначально верно
>
> *и если* выполняется действие типа A
>
> *то* свойство Q становится верным после этого выполнения.

Такое предложение будет также обозначаться сокращенно:

{P} A {Q}

Это соответствует нашему уже соблюдаемому соглашению, по которому комментарии, заключенные в фигурные скобки, выражают, насколько это возможно, «**утверждения**» (или «**высказывания**»), всегда справедливые при нахождении активной точки в заданном месте программы.

Свойства P и Q в этих примерах будут утверждениями, касающимися переменных программы. Некоторые из этих утверждений тривиальны. Так, если объявлено

переменная i : ЦЕЛАЯ

то известно, что на протяжении всей программы переменная i будет иметь соответствующее значение, которое является элементом из N. Другие утверждения могут зависеть от присваиваний, в которых участвуют переменные. Так, мы познакомились в гл. II с фундаментальным свойством присваивания: Если P – произвольное утверждение, то

для того чтобы P было верным

после выполнения x ← E (где E – выражение)

достаточно, чтобы P [E → x] (т.е. свойство P после замены всех вхождений x на E) было верным изначально

Например:

{x > n} x ← x + 1 {x > n + 1}

т.е. если x > n и если выполняется присваивание x ← x+1, то x > n + 1 после этого. В самом деле, P[E → x] есть в этом случае замена x + 1 на x в {x > n + 1}, т.е. {x + 1 > n + 1}, что эквивалентно {x > n}. Другой пример:

{0 < x < 1} x ← 1/x + y {x > y + 1}

Действительно, если P есть {x > y + 1}, то P[1/x + y → x] есть 1/x + y > y + 1, или 0 < x < 1 ⇒ 1/x + y + 1

Заметьте, что рассуждения ведутся в *обратном* направлении (от конечного утверждения к начальному утверждению). Это соответствует естественной ситуации, когда надо достичь некоторого «заданного» состояния программы, и можно строить начальные гипотезы и последующие операторы в зависимости от этого заключительного утверждения.

Часто встречающийся тип заключительного утверждения такой: «некоторая переменная,

соответствующая разыскиваемому результату, имеет своим значением переменную функцию других переменных, соответствующих данным этой задачи». В качестве примера можно применить показанные ниже свойства для доказательства правильности программы из упражнения III.8, которая дает способ вычисления A^B. Заключительным утверждением здесь является $\{z = A^B\}$, где A и B – данные, а z – результирующая переменная.

III.4.2. Свойства цепочки

Свойства *цепочки* очень просты. Они обозначаются («отношение Шаля») в виде

если {P} A {Q} *и* {Q} B {R}
то {P} A; B {R}

т.е. если A таково, что его выполнение при верном {P} приводит к ситуации, в которой {Q} верно, и таким же образом выполнение B при верном {Q} приводит к ситуации, в которой {R} верно, то выполнение B вслед за A, отправляясь от исходной ситуации, в которой верно {P}, приведет к конечной ситуации, где {R} верно (уф!).

Пример: Предположим, что числа x и y таковы, что $\{0 < x < 1\}$.

Тогда, если выполнить

$x \leftarrow 1/x + y;\ x \leftarrow x + 1$

то получим $\{x > y + 2\}$. Действительно, мы видели, что

$\{0 < x < 1\}\ x \leftarrow 1/x + y\ \{x > y + 1\}$

и что $\{x > n\}\ x \leftarrow x + 1\ \{x > n + 1\}$

III.4.3. Свойства альтернативы

Свойства альтернативы записываются так:

если {P **и** C} A {Q}
и {P **и** ~C} B {Q}
то {P} **если** C **то** A **иначе** B {Q}

Это правило выражает простой факт, что при выполнении альтернативы **если C то A иначе B** дедуктивные свойства A или B применяются в зависимости от того, истинно C или ложно.

Пример: покажем, что

$\{x < y\}$
если $x \leftarrow 2$ **то**
$y \leftarrow x^2$
$x \leftarrow -x$
иначе $y \leftarrow x + y + 3$
$\{y > x + 1\}$

(Очевидно ли это априори?)

По свойствам альтернативы достаточно показать отдельно два положения:

а) $\{x<y$ **и** $x\le -2\}\ y \leftarrow x^2;\ x \leftarrow -x\ \{y > x + 1\}$

б) $\{x<y$ **и** $x\ge -2\}\ y \leftarrow x + y + 3\ \{y > x + 1\}$

Чтобы доказать **а)**, нужно применить правила присваивания и цепочки. Рассуждая *справа налево,* мы сможем показать (применяя правила присваивания к $x \leftarrow -x$ и $P = \{y > x + 1\}$), что

$\{x < y$ **и** $x < -2\}\ y \leftarrow x^2\ \{y > -x + 1\}$

т.е. показать, что (заменой y на x^2 в {y > –x + 1})
{x < y **и** x < –2} ⇒ {x^2 > –x + 1}
что верно, так как x < –2 ⇒ x^2 > –x + 1

Чтобы доказать **б)**, достаточно показать, что (заменой y на x + y + 3 в {y > x + 1})
{x < y **и** x≥ –2}⇒{x + y + 3>x+1}
что верно, поскольку
{x < y **и** x > –2}⇒{y > –2}

III.4.4. Свойства цикла

Свойства цикла типа **пока** записываются в виде

а) **пока** C **повторять** A {~C}

(т.е. на выходе из цикла **пока** условие цикла всегда ложно). Это соответствует естественному использованию цикла **пока:** нужно обеспечить в некоторой точке программы правильность некоторого утверждения D. Известно действие A (которое, очевидно, может быть сложным), от которого ожидают, что оно приближает начальное состояние к состоянию, где D истинно. Полагая C = ~D (обратное условие), надо **повторять A, пока** C будет **истиной**

б) *если* {P **и** C} **A** {P}
то {P} **пока** C **повторять** A {P}

Это свойство исключительно важно. Оно выражает тот факт, что если {P} **инвариантно** по отношению к действию A т.е. если выполнение A оставляет {P} истинным при исходно верном {P} во всяком случае когда условие C верно, то {P} тоже инвариантно но отношению к циклу

пока C **повторять** A

Понятие инварианта цикла, смысл которого не всегда очевиден при первом чтении, играет решающую роль в построении программ систематическими методами. В самом деле, можно считать, что цикл **пока** определен своим условием окончания и инвариантом. Всегда, когда это возможно, мы будем указывать одновременно с циклом его инвариант. Упражнение III.9 показывает применение инварианта к доказательству правильности нетривиальной программы.

Инвариант часто предстает в форме: «такая–то переменная» (программы) «имеет в качестве значения такую–то величину» (связанную с решаемой задачей). В качестве примера ниже рассмотрена простая программа, которая определяет, есть ли в массиве T[m:n] значение x:

```
переменные i: ЦЕЛАЯ,
           есть: ЛОГИЧЕСКАЯ;
i ← m; есть ← ложь;
{поиск x в массиве T[m: n]}
пока i ≤ n и ~есть повторять
    есть ← (T[i] = x);
    i ← i + 1
```

Оставляем читателю доказательство того, что свойство P:

{Переменная «есть» будет иметь значение **истина** тогда и только тогда, когда i > m,

T[I – 1] = x и для всех j, содержащихся между m и i – 2 включительно, T[j] ≠ x}

или более формально:

{есть ⇔ I > m **и** T[I – 1] = x **и** (∀ j | m ≤ j ≤ I – 2, T[j] ≠ x)}[1]

является инвариантом по отношению к телу цикла.

Будучи верным сначала (поскольку есть = ложь и i = m), этот инвариант остается верным и на выходе из цикла. Соединенный с отрицанием условия цикла (т.е. {i > n **или** есть}), он дает в качестве окончательного высказывания

{есть = **ложь и** никакой элемент T не равен x,
или есть = **истина и** x = T[i – 1]}

что и выражает требуемый результат.

Другую, более простую версию этой программы можно записать в виде

```
{поиск x в массиве T[m : n]}
i ← m
пока i ≤ m и T[i] ≠ x повторять
    | i ← i + 1
```

Предоставим читателю найти соответствующий инвариант и доказать, что в конце этой программы можно утверждать, что

{i > n и никакой элемент T не равен x,
или i ≤ n и x = T[i]}

Замечание: правильное выполнение этой программы требует, чтобы второй терм условия **пока**, т.е. T[i]≠x, проверялся только при условии, что первый имеет значение **истина**. Действительно, в противном случае можно было бы пытаться при i > n вычислять T[i], которое не определено. Это требование выдерживается по определению логического **и** (*AND* в АЛГОЛе W (III.5.3.1)).

Важно отметить одно ограничение отмеченных выше свойств цикла. Мы всегда предполагали, что можно рассматривать правильные высказывания после цикла, т.е. когда его выполнение **закончилось**. Это не всегда имеет место: легко построить примеры некончающихся циклов. Каждому программисту доводилось написать, по крайней мере однажды, программу, которая давала невольную реализацию «бесконечного цикла», упоминавшегося в начале главы. Если такая ситуация не была целью, то надо *доказывать,* что цикл завершается. Для этого надо выделить некоторую связанную с циклом целую неотрицательную величину m, зависящую от переменных программы и называемую *управляющей величиной* (более точно, доказывают, что свойство , {m ≥ 0} есть инвариант цикла и что оно исходно верно), и показать, что каждое выполнение цикла уменьшает m по крайней мере на единицу.

В некоторых практических примерах достаточно легко найти управляющую величину и доказать, что цикл завершается (так, читателю предлагается доказать, что n – i + 1 есть управляющая величина для двух вышеприведенных программ). Тогда без всяких сложностей можно пользоваться другими свойствами управляющих структур.

III.4.5. Заключение

Важно отметить (оставляя читателю возможность поразмышлять о философской стороне этого результата), что не существует никакого общего метода, который несомненным образом доказывал бы завершимость произвольной программы. Хуже того, *не может существовать никакого алгоритмического метода,* который по внешнему виду произвольной программы p и произвольных данных d был бы способен решить, завершится ли программа p, примененная к данным d (см. упражнение III.11).

[1] Символ "⇔" читается «эквивалентно», а символ ∀ – это квантор общности, «для всех»: $\forall_j$ «для всех j, таких, что …»

В конкретных практических случаях, однако, этот вопрос часто разрешим. Мы вернемся к такой же проблеме в связи с завершением рекурсии (гл. VI).

Чтобы закончить этот, может быть, немного более абстрактный, чем другие, раздел, мы напомним еще раз тот факт, что введенные нами базовые структуры оставляют программисту возможность рассуждать статически о его программе и *доказывать* правильность относящихся к ней **утверждений.** Свойства этих структур дают естественную основу при написании программы, *исходя* каждый раз *из определенных утверждений:* речь идет о том, чтобы писать известные операторы, исходя из известных гипотез, а не выписывать строчки кодов, от которых ожидается, что они приведут к приемлемому результату. Мы покажем многочисленные примеры построения программы на основе утверждений, в частности в случае, когда очень легко ошибиться, если не использовать этот метод (дихотомический поиск, VII.2.3). Более подробно метод исследуется в гл. VIII**,** где мы обсудим практический смысл методов «доказательства программ» и возможности их применения в повседневном программировании (VIII.4.2.2).

Как бы ни были несовершенны эти приемы, они представляют большой шаг в превращении программирования в настоящую науку.

III.5. Представление управляющих структур в ФОРТРАНе, АЛГОЛе W и ПЛ/1

Никакой язык программирования не предлагает непосредственно все структуры, описанные в разд. III.3. Большая часть языков, однако, располагая некоторыми структурами, позволяет моделировать другие. Посмотрим, что в связи с этим предлагают ФОРТРАН, АЛГОЛ W и ПЛ/1.

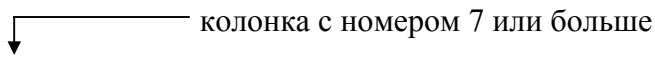
Нам будет удобнее рассмотреть здесь эти структуры в другом порядке: цепочка, цикл, ветвление. Подпрограммы подробно обсуждаются в гл. IV.

III.5.1. Цепочка

Во всех языках существуют средства заставить следовать один оператор за другим.

- **В ФОРТРАНе** для этого достаточно начать новую строку: во всех случаях, кроме карт–продолжений (гл. II), переход к новой строке является разделителем операторов и потому управляющей структурой. Например, последовательность операторов

ФОРТРАН




```
I = 3
J = 5
IPLUSJ = I + J
```

направит в *IPLUSJ* (переменная, предполагаемая целой, так же как *I* и *J*) значение 8.

- В **АЛГОЛе W** отделяют один оператор от другого, выполняемого вслед за ним, с помощью точки с запятой:

АЛГОЛ W

```
I := 3;
J :=5;
IPLUSJ := I + J
```

(знак := указывает присваивание справа налево в отличие от знака =, который является знаком оператора отношения, встречающегося в логических выражениях). Напомним, что формат языка свободный, т.е. последовательные операторы могут размещаться в одной строке.

- **ПЛ/1** заимствовал из АЛГОЛа точку с запятой, но использует его более систематически: всякий оператор сопровождается точкой с запятой, которая поэтому является не разделителем операторов, а признаком конца оператора или объявления. Следовательно, в ПЛ/1 (где формат тоже свободный) пишут

ПЛ/1

```
I = 3;
J = 5;
IPLUSJ = I + J;
```

(знак = в ПЛ/1 одновременно и знак присваивания, и знак оператора отношения).

Перечисленные выше конструкции, хотя и позволяют эффективно описать последовательность выполняемых операторов, тем не менее не реализуют цепочку так, как мы ее задумали, т.е. как сложное действие, но в целом эквивалентное одному–единственному оператору. Единственное средство достичь такой цели в ФОРТРАНе – это написать подпрограмму; к этому мы еще вернемся в следующей главе. АЛГОЛ же ввел важное понятие блока. Блок в АЛГОЛе W есть последовательность нуля, одного или более операторов, которые отделены друг от друга точкой с запятой, при этом вся последовательность окаймляется символами *BEGIN* и *END,* играющими роль скобок. Пример блока:

АЛГОЛ W

```
BEGIN
I :=3;
J := 5;
IPLUSJ := I + J
END
```

Отметим, что знак точка с запятой играет синтаксически роль разделителя, а не указателя конца оператора. Поэтому не является необходимым включение этого символа после *BEGIN* и перед *END*. Можно сравнить роль *BEGIN, END* и точки с запятой с открывающей скобкой, закрывающей скобкой и запятой соответственно в обычном математическом выражении

f(x, y, z)

Такой блок, как описан выше, синтаксически эквивалентен единственному оператору и может появляться всюду, где разрешен один оператор. В частности, один из операторов, составляющих блок, может в свою очередь быть блоком; так, вполне допустима следующая конструкция, если были правильно объявлены переменные *I, J, K, X, Y, U, V:*

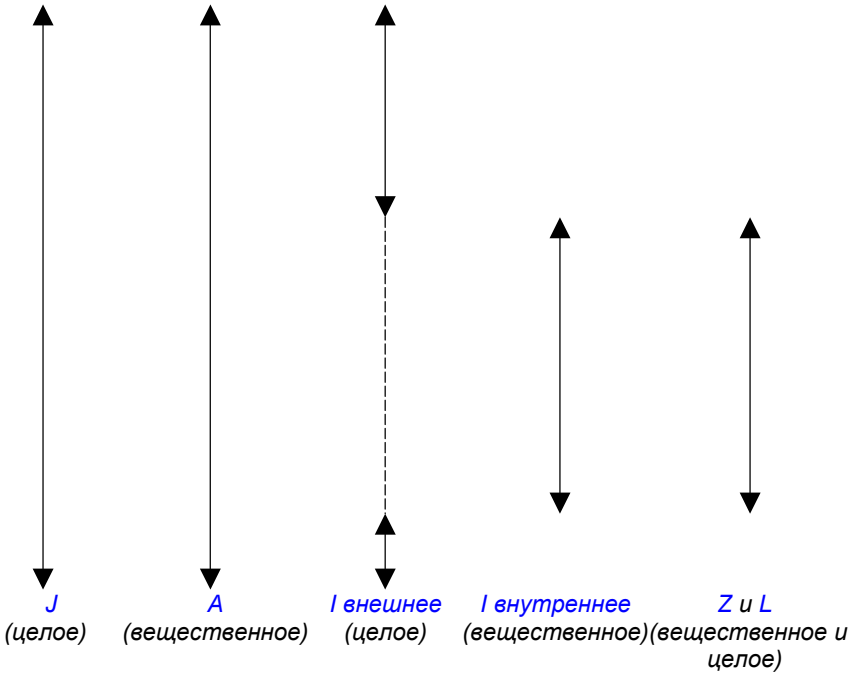
АЛГОЛ W

```
BEGIN
WHILE I < 0 DO
        BEGIN I := I + J;
        IF J > 0 THEN
                BEGIN
                X := Y;
                Z := K
                END
        ELSE
                U := V
        END;
I := I + 1
END
```

Понятие блока, введенное АЛГОЛом, сопровождается другим важным понятием: **область действия идентификатора.** Идентификатор, связанный с переменной, может быть объявлен внутри блока (в каждом блоке все объявления должны предшествовать операторам); это относится ко всем переменным программы АЛГОЛа W, потому что программа есть не что иное, как блок (сопровождаемый точкой). С переменной можно работать только внутри блока, в котором объявлен ее идентификатор, и в блоках, в него включенных. Идентификаторы, объявленные одинаковыми именами в двух разных блоках (разделенных или включенных один в другой), представляют переменные, которые никак не связаны между собой.

Вот пример программы на АЛГОЛе W (который не претендует на описание содержательного вычисления):

```
        BEGIN
        INTEGER I, J;
        REAL A;
(1)     I := 2;
        A := 3.2;
(2)     J := I * I;
        IF J = I + 1 THEN
            BEGIN
            INTEGER L;
            REAL I, Z;
(3)         I := 3.5;
(4)         L := J + 2;
            Z := 0.0
            END;
(5)     I := 5; J := I
        END
```



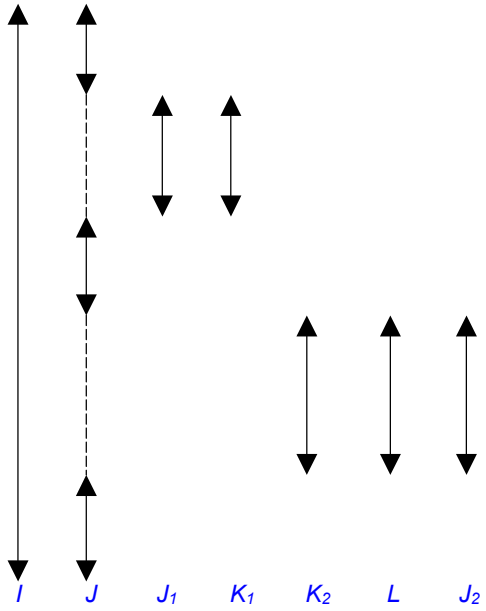
Стрелки справа от программы указывают область действия каждого идентификатора, т.е. ту часть программы, в которой разрешено его упоминать. Две переменных носят одно и то же имя *I*: одна – целая во внешнем блоке, другая – вещественная, во внутреннем блоке.

В операторах (1), (2) и (5) участвует *«I внешняя»*; в (3) речь идет об *«I*

внутренней»; *I внешняя* не упоминается во внутреннем блоке, потому что другая локальная переменная этого блока носит то же имя. Напротив, (4) содержит ссылку на *J*: всякая объявленная в некотором блоке переменная доступна во внутренних по отношению к нему блоках, если они не содержат объявления переменной с тем же именем. Переменные *Z* и *L,* так же как *I* внутренняя, являются локальными переменными внутреннего блока; они не существуют за его пределами.

Другой пример:

```
BEGIN INTEGER I, J;
...
        BEGIN INTEGER J, K
        ...
        END;
        ...
        ...
        ...
        BEGIN INTEGER J, K, L;
        ...
        ...
        END;
...
...
END
```



Структура блока, такая, как она определена в сообщении об АЛГОЛе 60 [Наур 63], включает другое понятие, к которому мы вернемся в IV.6 и VI.3.4: это *динамическое распределение* памяти, которое порождает управление памятью с помощью *стека* – области памяти, занятой переменными некоторого блока и выделяемой только в начале каждого выполнения этого блока. Эта область, следовательно, может иметь переменную длину, что делает допустимыми, например, массивы с границами, фиксируемыми только в начале выполнения блока, в котором они объявлены, или подпрограммы, которые вызывают сами себя (рекурсия).

В вопросе о блочном структурировании программ ПЛ/1 воспринял идеи АЛГОЛа. ПЛ/1 делает отличие между тремя видами блоков:

- блоки типа
  DO; оператор 1;
  ...
  оператор n; END;

(или «группы *DO*» в терминологии ПЛ/1), которые являются простыми цепочками операторов и не могут иметь локальных переменных ;

- блоки типа
  BEGIN;
  оператор 1;
  ...
  оператор n; END;

которые могут содержать объявления переменных;

- блоки типа
  PROCEDURE;
  оператор 1;
  ...
  оператор n; END;

которые также могут обладать локальными переменными; речь идет о блоках, соответствующих подпрограммам, которые изучаются в гл. IV.

Заметим, что в ПЛ/1 слова *BEGIN, DO, PROCEDURE, END* рассматриваются не как ограничители, а как настоящие «операторы» и всегда сопровождаются точкой с запятой (для *DO* и *PROCEDURE* после возможных дополнительных спецификаций); точка с запятой играет здесь в отличие от соглашений АЛГОЛа роль обязательного *указателя конца* оператора или объявления.

III.5.2. Циклы

III.5.2.1. Бесконечный цикл

Чтобы смоделировать цикл

повторять бесконечно
| действие

можно использовать в наших трех языках оператор передачи управления *GOTO* («идти к»), приказывающий продолжать выполнение программы с оператора, обозначенного меткой:

метка начало *действия*
продолжение *действия GOTO метка*

• В ФОРТРАНе *метка* – это десятичное число, содержащее от одной до 5 цифр; метка записывается в колонках с первой по пятую той строки, которая содержит первый оператор *действия*. Например,

ФОРТРАН

```
100 I = 5
    GOTO 100
```

будет неопределенно долго повторять присваивание значения 5 переменной *I* (мало интересная программа).

• В АЛГОЛе W и ПЛ/1 метка – такой же идентификатор, как и всякий другой (т.е. буква, за которой, возможно, следуют буквы или цифры), но, однако, не объявленный явно; метка как бы объявляется контекстом: она отмечает оператор, т.е. сопровождается двоеточием и ставится перед этим оператором[1]:

АЛГОЛ W

```
METKA: I: =5;
GOTO METKA
```

ПЛ/1

```
METKA: I = 5;
GOTO METKA;
```

В этих двух последних языках можно задать бесконечное повторение также с помощью цикла **пока** ... **повторять**. В ПЛ/1 можно использовать конкретную форму «группы *DO»;* см. разд. III.5.2.3.в.

[1] В ПЛ/1 метки можно объявлять и явно:
DECLARE METKA LABEL;
Это используется особенно в массивах меток (см. III.5.3.2).

III.5.2.2. Циклы пока и до

Цикл **пока** ... **повторять** записывается в АЛГОЛе W в виде

WHILE условие DO
оператор

где *условие* – логическое выражение, определенное в гл. II, а *оператор* – любой оператор языка (в частности, он может быть блоком). Как уже говорилось, мы будем размещать структуру цикла при написании программы, **сдвинув** вправо *оператор* на несколько позиции литер; блок, внутренний по отношению к предыдущему, будет снова сдвинут на несколько дополнительных позиций и т.д.

В ФОРТРАНе, чтобы воспроизвести цикл **пока** ... **повторять**, применяют логическую проверку и затем *GOTO:*

```
100 IF (обратное условие) GOTO 200
        текст оператора
        GOTO 100
200 продолжение программы
```

обратное условие – это **логическое выражение;** предполагается, что *100* и *200* – номера меток, которые не участвуют в программе. Здесь *обратное условие* представляет собой условие, отрицающее использованное в только что приведенном представлении АЛГОЛа W; его можно получить из условия АЛГОЛа W, например, с помощью логической операции *NOT*.

Здесь тоже мы будем использовать сдвиг вправо при изображении структуры цикла; напомним, что это всегда можно сделать, так как обычно пробелы в ФОРТРАНе не значащие. Так как используемая в ФОРТРАНе конструкция со всей очевидностью представляет более низкий концептуальный уровень, нежели конструкция **пока**, мы будем всегда напоминать ее содержание с помощью предшествующего комментария:

ФОРТРАН

```
            I = 1
C           / ПОКА A(I)>0 ПОВТОРЯТЬ /
100         IF (A(I).LE. 0) GOTO 200
                    I = I + 1
                    GOTO 100
200         …
```

Цикл **пока** ПЛ/1**,** который мы изучим подробнее немного позднее, записывается в виде

```
DO WHILE (условие);
        действие 1;
        ...
        действие n;
        END
```

И здесь в физическом размещении программы видна попытка отразить ее динамическую структуру.

Никакой из наших трех языков не располагает таким приемом:

повторять
| действие **до** условие

В АЛГОЛе W можно написать

действие;
WHILE ¬условие DO действие

(¬ – это операция АЛГОЛа W, которая позволяет ввести условие, обратное по

отношению к *условию*). Этот же метод непосредственно распространяется и на ПЛ/1.

В ФОРТРАНе требуемую конструкцию записывают, устанавливая «вручную» проверку в конце цикла:

```
100 начало текста действия
    продолжение текста (если необходимо)
    ...
    IF (обратное условие) GOTO 100
```

где *обратное условие* – это условие, отрицающее *условие* (выраженное, например, с помощью операции *.NOT.*). Эту запись можно, конечно, перенести в ПЛ/1 и в АЛГОЛ W.

III.5.2.3. Цикл со счетчиком

Три наших языка предлагают цикл со счетчиком, в котором можно специфицировать *шаг:*

а) *АЛГОЛ W*

В АЛГОЛе W оператор *FOR* имеет вид

```
FOR счетчик := начзначение STEP шаг UNTIL предел DO
                оператор
```

где *счетчик* – *это* идентификатор (объявленный неявно, см. ниже), *начзначение, шаг* и *предел* – целые выражения, *оператор* – произвольный оператор языка, *STEP шаг* может быть опущено, если *шаг* = 1. Он эквивалентен:

- если *начзначение* ≤ *предел* и *шаг* ≥ *0* то конструкции

```
BEGIN INTEGER СЧЕТЧИК, НАЧЗНАЧЕНИЕ, ПРЕДЕЛ, ШАГ;
   НАЧЗНАЧЕНИЕ := начзначение; ШАГ:= шаг; ПРЕДЕЛ := предел;
   СЧЕТЧИК : = НАЧЗНАЧЕНИЕ;
   WHILE СЧЕТЧИК < = ПРЕДЕЛ DO
   BEGIN
         оператор;
         СЧЕТЧИК : = СЧЕТЧИК + ШАГ
   END
END
```

- если *начзначение* ≥ *предел* и *шаг* ≤ *0*, то конструкции

```
BEGIN INTEGER СЧЕТЧИК, НАЧЗНАЧЕНИЕ, ПРЕДЕЛ, ШАГ;
   НАЧЗНАЧЕНИЕ: = начзначение; ШАГ: = шаг; ПРЕДЕЛ : = предел;
   СЧЕТЧИК : = НАЧЗНАЧЕНИЕ;
   WHILE СЧЕТЧИК >= ПРЕДЕЛ DO
   BEGIN
         оператор;
         СЧЕТЧИК: =  СЧЕТЧИК + ШАГ
   END
END
```

- если *предел – начзначение* и *шаг* имеют разные знаки, то «пустому» действию.

В указанных «эквивалентностях» объявления локальных переменных *НАЧЗНАЧЕНИЕ, ШАГ, ПРЕДЕЛ* и их инициализация значениями *начзначение, предел, шаг* соответствуют тому, что в АЛГОЛе W целые выражения, определяющие начальное значение, шаг и границу цикла *FOR,* вычисляются единственный раз при входе в цикл, число выполнений которого не зависит от последующего изменения переменных, входящих в эти выражения (кроме случая *GOTO* вне цикла, практически дозволенного в АЛГОЛе W).

Резюмирующий пример: конструкция

АЛГОЛ W

```
FOR I : = M STEP N + 2 UNTIL P – 3 DO
          BEGIN
          M := N * P + 2;
          A(M) := A(I)+ 1
          END
```

где целые переменные *M, N, P* и массив *A* предполагаются ранее объявленными, эквивалентна по своему действию[1] конструкции

АЛГОЛ W

```
BEGIN
COMMENT VALEURINIT – НАЧЗНАЧЕНИЕ,
            LIMITE –ПРЕДЕЛ, PAS – ШАГ;
INTEGER I, VALEURINIT, LIMITE, PAS;
VALEURINIT := M; LIMITE := P – 3; PAS := N + 2;
I := VALEURINIT;
WHILE PAS*(LIMITE–I) >= 0 DO
      BEGIN
      M := N*P + 2;
      A (M): = A(I) + 1; I := I + PAS
      END
END
```

Умножение *PAS*(LIMITE – I)* было здесь введено только для того, чтобы проверить, имеют ли *PAS* и *LIMITE – I* одинаковые знаки, т.е. не произошел ли выход за границу (верхнюю или нижнюю в зависимости от ситуации). Разумеется, машинное представление цикла *FOR* не обязательно должно выполнять умножение, но может делать, например, проверку знаков.

Заметим (как говорилось уже в наших «эквивалентностях»), что счетчик цикла (*I* в нашем примере) рассматривается как его локальная переменная. Он, однако, не был объявлен (в действительности всякая ранее объявленная переменная с тем же именем была бы недопустима в теле цикла), и его значение теряется при выходе из цикла. Если желательно сохранить последнее значение счетчика (здесь *J*) перед выходом, надо включить в тело цикла присваивание типа

J := I

где *J* – некоторая объявленная переменная.

В теле цикла запрещено употребление операторов, которые могли бы модифицировать значение счетчика (присваивание *I*, например).

б) *ФОРТРАН*

Цикл со счетчиком записывается в **ФОРТРАНе** в виде

```
DO метка счетчик=начзначение, предел, шаг
    тело цикла
метка    последний оператор
```

колонки с 1 по 5

[1] Пренебрегая возможностью переполнения при умножении.

счетчик – это целая переменная; *начзначение, предел, шаг,* которые в соответствии со стандартом ФОРТРАНа должны быть целыми положительными переменными или константами, являются соответственно начальным значением, границей, сравниваемой со счетчиком, и шагом *шаг* может отсутствовать; в этом случае он принимается равным единице); *метка* – это обычная фортрановская метка, т.е. целое, содержащее от 1 до 5 цифр. Обычная практика рекомендует включать в качестве *последнего оператора* специальный оператор *CONTINUE;* это «пустой» оператор, который не выполняет никакого действия, но позволяет четко отделить тело цикла от управляющих операторов, как *END* в ПЛ/1. Применение *CONTINUE* в качестве *последнего оператора* необходимо, когда тело цикла заканчивается альтернативой (разд. III.3.2.1). Как и раньше, текст тела цикла (до *CONTINUE* включительно) будет отмечаться смещением.

В отличие от цикла *FOR* АЛГОЛа W, который был циклом типа пока, цикл *DO* в ФОРТРАНе – это цикл **повторять** ... **до**, т.е. тело цикла выполняется всегда по крайней мере один раз; сравнение *счетчика* с *пределом* выполняется в конце цикла; это может вызвать затруднение в случае, когда *начзначение* больше *предела:* тогда было бы логично рассматривать цикл *DO* как пустой оператор[1].

Это свойство ФОРТРАНа несколько стеснительно, и, поскольку цикл *FOR* должен уметь ни разу не выполняться при границах *предел* < *начзначение,* мы будем его трактовать как цикл **пока** (ср. выше) с соответствующими комментариями.

Как ив АЛГОЛе W, запрещено менять счетчик в теле цикла *DO* ФОРТРАНа.

Когда имеется несколько вложенных циклов (что разрешается без всяких ограничений в ФОРТРАНе; каждый транслятор устанавливает, однако, некоторый лимит на число уровней вложенности), *последний оператор* одного или нескольких из них и, следовательно, *метка* соответствующего *DO* могут быть общими, как показывает следующий пример:

ФОРТРАН

```
      INTEGER A (10), B (10, 10), C (10, 10, 10), D(10, 10, 10)
      INTEGER I, J, K
C     ЦИКЛ 1
      DO 100 J = 1,10
          A(I) = 1
C     ЦИКЛ 2
      DO 100 J = 1,10
          B(I, J) = 1
C     ЦИКЛ 3
      DO 200 K = 1,10
          C (I, J, K) = 1
200       CONTINUE
C     ЦИКЛ 4
      DO 100 K = 1,10
          D (I, J, K) = 1
100   CONTINUE
```

В этом примере будут выполнены присваивания 10 элементам из *A,* 100 элементам из *B,* 1000 элементам из *C* и *D*; последняя строка *100 CONTINUE* служит последним оператором циклов 1, 2 и 4. Незачем говорить, что такая возможность не рекомендуется тому, кто хочет писать хорошо понятные программы, и что предпочтительнее связать с каждым циклом отдельное завершающее

[1] Это свойство не является, собственно говоря, характеристикой языка, официальный стандарт которого довольствуется неопределенностью действия цикла при *начзначение > предел.* Но почти все трансляторы используют эту свободу только частично, чтобы разместить проверку в конце цикла, т.е. чтобы реализовать цикл **повторять** ... **до**.

CONTINUE (и, значит, различные метки).

в) *ПЛ/1*

Циклы в ПЛ/1 представляют частный случай «группы *DO*» (т.е. напомним, «блок», который группирует операторы, но не имеет локальных идентификаторов). Это, как мы увидим, может приводить к неприятностям.

Цикл со счетчиком (типа **пока**) можно записать в виде

```
DO счетчик = начзначение TO предел BY шаг;
        тело цикла;
        END;
```

Порядок спецификаций *TO* и *BY* безразличен. Если *BY шаг* опущено, то шаг принимается равным единице; если опущено *TO предел,* то имеет место «бесконечный цикл» при условии, что спецификация *BY* все же имеется; в противном случае это была бы простая «группа *DO»,* выполняемая один–единственный раз с *начзначение* в качестве значения переменной *счетчик.*

Как и в АЛГОЛе W, проверка продолжения цикла эквивалентна

*шаг * (предел–счетчик) ≥ 0*

Примеры:

DO I = 1 TO 10 BY 5;	выполняется 2 раза (*I* = 1, затем 6)
DO I = 1 TO 10;	выполняется 10 раз
DO I = 1 BY 5;	бесконечный цикл (*I* = 1, 6, 11,...)
DO I =1;	выполняется 1 раз

начзначение, предел, шаг – скалярные выражения; область действия *I* распространяется на всю **процедуру,** в которой участвует цикл (понятие процедуры, используемой ПЛ/1 для «программ» и «подпрограмм», будет уточнено в гл. IV). После выхода из цикла *I* сохраняет то значение, которое было причиной этого выхода (ср. программу задачи III.2).

ПЛ/1 располагает циклом **пока**:

```
DO WHILE (логическое выражение); тело цикла;
        END;
```

Можно комбинировать управление с помощью счетчика и управление с помощью условного выражения:

```
DO счетчик = начзначение TO предел BY шаг WHILE (условие) тело цикла; END
```

Эта запись эквивалентна следующей:

```
счетчик = начзначение;
DO WHILE (условие & шаг * (предел–счетчик) >=0));
        тело цикла;
        счетчик = счетчик + шаг;
        END;
```

Примеры:

*DO I = 1 TO 10 BY 2 WHILE (I**2 < 50);*	выполняется 4 раза (*I* = 1, 3, 5, 7);
DO I = 1 WHILE (условие);	выполняется 0 или 1 раз
DO I = 1 BY 5;	бесконечный цикл
DO I = 1 BY 5 WHILE (условие);	повторяется, пока *условие* истинно (быть может, бесконечно).

Синтаксически ПЛ/1 располагает правилами, позволяющими «закрыть» одновременно несколько *DO* с помощью одного *END,* Еще больше, чем в ФОРТРАНе, эти правила приводят к неприятностям в программах, и мы сбросим завесу

стыдливости с этого сомнительного правила ПЛ/1.

г) *Другие языки*

Чтобы закончить этот обзор циклов, упомянем интересную конструкцию АЛГОЛа 68. В АЛГОЛе 68 самая общая конструкция цикла имеет вид

для *счетчик* **от** *начзначение* **шаг** *р* **до** *предел* **пока** *условие*
цк *оператор 1; оператор 2,... оператор n* **кц**

цк (цикл) и **кц** (конец цикла) являются «скобками», которые выделяют совокупность повторяемых операторов.

Можно опустить все или часть указаний, которые предшествуют этим «скобкам»:

опущение	вариант по умолчанию
шаг р	р = 1
для *счетчик*	*счетчик* не должен использоваться в *операторе*
от *начзначение*	*начзначение* = 1
до *предел*	*предел* = ±∞ (цикл может завершиться только оператором **пока** *условие* или явным ветвлением)
пока *условие*	*условие* = *истина* (всегда верно)

Примеры (в которых предполагается, что тело цикла не содержит операторов перехода *идти к,* ведущих во вне цикла):

пока *С* **цк** ... **кц**	выполняется, пока *С* имеет значение **истина**
цк ... **кц**	бесконечный цикл
от *10* **до** *30* **цк** ... **кц**	выполняется 21 раз; значение индекса недоступно в теле цикла
до *30* **цк** ... **кц**	выполняется 30 раз

III.5.3. Ветвления

III.5.3.1. Альтернатива

а) *АЛГОЛ W*

В АЛГОЛе W существует как таковой переключатель, имеющий две ветви, **если** ... **то** ... **иначе**:

```
IF логическое выражение THEN
        оператор 1
ELSE                                                    (1)
        оператор 2
```

Можно также опустить часть «**иначе** ...»;

```
IF логическое выражение THEN                            (2)
        оператор 1
```

В форме (1) *оператор 1* не может быть произвольным оператором. Если допустить полную свободу, то можно было бы встретиться с такой конструкцией:

```
IF X = Y THEN
    IF Z = T THEN
        IF U = V THEN
            A := B
        ELSE C := D
    ELSE E := F
```

где невозможно однозначно определить, к каким *THEN* относятся два *ELSE*. Из этих соображений АЛГОЛ W запрещает в качестве *оператора 1* в (1) условные операторы, такие, как циклы, *CASE* и *ASSERT* (см. гл. VIII). С другой стороны, блок разрешен всегда, что позволяет писать на этот раз без двусмысленностей, например:

АЛГОЛ W

```
IF X = Y THEN
        BEGIN IF Z = T THEN
                BEGIN IF U = V THEN
                        A := B
        END
ELSE C := D
END
          ELSE E := F
```

Конечно, оператор *оператор 2,* который следует за *ELSE,* произволен; АЛГОЛ W допускает также произвольный оператор после *THEN* в форме (2) (без «**иначе**...»); другими словами,

АЛГОЛ W

```
IF X = Y THEN
        IF Z = T THEN
                U := V
        ELSE W := Z
```

разрешено, причем *ELSE* относится к *IF Z = T*. Однако в АЛГОЛе W обычно рекомендуется всегда употреблять блок *BEGIN...END,* если желательно использовать условный оператор в части *THEN* альтернативы независимо от ее типа (1) или (2).

АЛГОЛ W обладает одним интересным свойством, которое касается, по правде говоря, условных выражений, а не альтернатив, но используется чаще всего именно в написании альтернатив (и циклов). Представим себе массив целых *TAB:*

```
INTEGER ARRAY TAB(1::N)
```

и пусть *I* – целое, о котором известно, что оно положительное. Часто случается ситуация, в которой желательно выполнить некоторое действие *A* после проверки некоторого связанного с *TAB(I)* условия, если *I* – допустимый для *TAB* индекс (например, *TAB(I)<0)*.

Тогда можно писать

```
IF (I <= N) AND (TAB(I) < 0) THEN
                A
```

не рискуя выйти за разрешенные границы изменения индекса при вычислении второго условия, когда *I > N;* в самом деле, определение АЛГОЛа W уточняет, что при вычислении логического выражения

a_1 *AND* a_2

a_2 не вычисляется, если a_1 есть *FALSE* (в этом случае результатом необходимо будет *FALSE)*. Точно так же

a_1 *OR* a_2

есть *TRUE,* если a_1 имеет такое значение; здесь тоже нет необходимости

вычислять a_2. Это соглашение позволяет:

- повысить качество рабочей программы, выдаваемой транслятором;
- облегчить, как мы видели, работу программиста.

Следует, однако, знать, что происходит в случае, когда a_2 является, например, вызовом логической процедуры (гл. IV), которая может изменить глобальные переменные.

б) *ПЛ/1*

ПЛ/1 тоже имеет альтернативу с явной или неявной **иначе**;

```
1) IF условное выражение THEN
       оператор 1; ELSE
       оператор 2;
2) IF условное выражение THEN
       оператор;
```

Можно организовать вложения *IF;* правило определяет, что каждое *ELSE* относится к последнему встретившемуся *THEN,* в сложной структуре вложенных *IF ... THEN ... ELSE* можно задать пустую часть **иначе**:

```
IF условие THEN оператор; ELSE;
```

в) *ФОРТРАН*

ФОРТРАН обладает ограниченным логическим условием, «логическим *IF»:*

```
IF (условие) оператор
```

оператор – это единственный оператор, который к тому же не может быть ни *DO,* ни логическим *IF.* Этот механизм предписывает выполнять *оператор,* если *условие* верно; в противном случае он эквивалентен пустому действию.

Чтобы перевести

```
если условие то
        | действие 1
иначе
        | действие 2
```

надо будет написать (всегда, кроме конкретного случая, в котором *действие 2* пустое, а *действие 1* содержит только один оператор)

```
      IF (условие) GOTO 100
         код
         для «действия 2»
         GOTO 200
100      код
         для «действия 1»
200      продолжение программы
```

Эта схема может слегка усложниться в случае, когда альтернатива является последним оператором тела цикла *DO;* в этом случае следует замкнуть обе ветви альтернативы на «пустой» оператор *CONTINUE* (разд. III.5.2.3.б), за которым будет следовать *«продолжение программы».* Так, фортрановский перевод конструкции

```
для i от m до n повторять
    | действие 1;
    | если условие то
    |       | действие 2
    | иначе
    |       | действие 3
```

будет иметь вид

```
        DO 200 I = M, N
        код для «действия 1»
        IF (.NOT. условие) GOTO 100
              код для «действия 2»
              GOTO 200
100     код для «действия 3»
200     CONTINUE
        продолжение программы
```

Альтернатива имеет простое представление в ФОРТРАНе в том частном случае, когда пишут

```
если с то
    | x ← e1
иначе
    | x ← e2
```

где x – переменная, а e_1, и e_2 – выражения, не содержащие ни x, ни вызовов подпрограммы, которые могут изменить состояние программы. В таком случае можно заменить альтернативу на

```
| x ← e2
| если с то x ← e1
```

т.е. в ФОРТРАНе

```
x = e2
IF (c) x = e1
```

Так можно уйти от обычных перекрестных передач управления.

Читатель, по–видимому, обратил внимание на смещения текста, которые мы используем по примеру различных языков программирования при изображении альтернатив и циклов, чтобы яснее выделить структуру.

III.5.3.2. Многозначные ветвления. Таблицы переходов

а) *CASE OF* в АЛГОЛе W

В частном случае конструкции **выбрать** (разд. III.2.2), когда n возможностей взаимно исключены, АЛГОЛ W предлагает оператор *CASE OF,* имеющий следующую форму:

```
CASE выр OF
      BEGIN
      оператор 1;
      оператор 2;
      оператор n
      END
```

выр – это выражение, а *оператор 1, ..., оператор n* – произвольные операторы (которые могут быть, например, блоками *CASE ... OF ...* и т.д.). Эта конструкция эквивалентна следующей:

```
BEGIN
COMMENT VAR – ПЕРЕМЕННАЯ, ERREUR–ОШИБКА;
INTEGER VAR; VAR: = выр;
            IF VAR = 1 THEN BEGIN оператор 1 END
            ELSE      IF VAR = 2 THEN BEGIN оператор 2 END
ELSE ...
ELSE ...
ELSE IF VAR = n THEN BEGIN оператор n END
ELSE ERREUR
```

END

(где *операторы i* «вставлены в блок» во избежание двусмысленности, которая могла бы быть вызвана перекрытием условных операторов; каждый *оператор i* может быть в свою очередь оператором *IF ... THEN ...* или *IF ... THEN ... ELSE ...*).

ERREUR – это вызов процедуры обработки ошибки; эта процедура, являющаяся частью системы, печатает адекватную диагностику и заканчивает выполнение программы. В обозначениях разд. III.3.2 вышеприведенный оператор *CASE OF* есть перевод на АЛГОЛ W конструкций

выбрать
    выр = 1: оператор 1,
    выр = 2: оператор 2,
    ...
    выр = n: оператор n
иначе ошибка

Если желателен иной оператор после иначе, чем диагностика стандартной ошибки, необходимо его явно программировать перед *CASE* в виде условного оператора

IF (выр < 1) OR (выр > n) THEN оператор 0
ELSE CASE выр OF
    BEGIN
    оператор 1;
    оператор 2;
    ...
    ...
    оператор n
    END

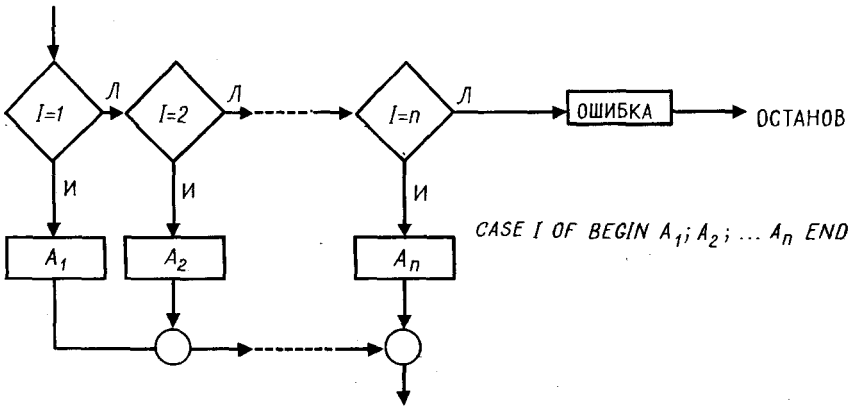
б) *индексированные переходы в ФОРТРАНе*

ФОРТРАН имеет оператор, который немного напоминает *CASE ... OF ...* : индексированное ветвление. Этот оператор записывается в виде

GOTO (метка1, метка2, ... меткаn), целзначение

где *целзначение* – целая переменная, а *метка1*, *метка2,..., меткаn* – номера меток в программе (например, *100, 200, ..., 900*). Такой оператор осуществляет переход к *метке1,* если *целзначение* = 1, к *метке2,* если *целзначение* = 2, ..., к *меткеn* если *целзначение* = *n*. По стандарту ФОРТРАНа результат неопределен, если *целзначение* < *1* или *целзначение* > *n;* соглашения в этом случае различны в разных трансляторах; одно из них состоит в приведении к 1 всех отрицательных значений и нуля и к *n* – значений, превосходящих *n*.

В отличие от *CASE OF* АЛГОЛа W, ветви которого сходятся (за исключением неуместных *GOTO*):



выполнение индексированного перехода в ФОРТРАНе никак не предрешает последующего поведения точки выполнения. Разумеется, индексированный переход позволяет представить структуру **выбрать** ..., сгруппировав и ветвей с помощью *GOTO* на одну общую метку; именно так мы его будем использовать.

ФОРТРАН является единственным из наших языков, предлагающим специальный оператор для выбора *трех возможностей:* оператор

IF (арифметическое выражение) метка1, метка2, метка3

обеспечит переход к меткам *метка1, метка2, метка3* в зависимости от того, является ли *арифметическое выражение* (которое может быть целым или вещественным) соответственно отрицательным, нулевым или положительным. Этот оператор достался в наследство от ФОРТРАНа II и от его реализации на ИБМ 704, где он строго соответствовал машинной команде. Опыт показывает, что в большинстве случаев этот оператор используется в «сокращенной» форме, где две метки одинаковы (выбор среди двух ветвей). «Логический *IF*» ФОРТРАНа IV предлагает в этом случае более ясное и легко читаемое написание (см. задачу III.1). Кроме того, общие условия применения «арифметического» *IF* приводят часто, когда надо сравнить два числа, к сравнению их разности с нулем, что опасно в большинстве вычислительных машин из–за риска переполнения.

в) *массивы меток в ПЛ/1*

В ПЛ/1 не существует оператора типа *CASE OF*. Его можно было бы смоделировать:

- либо с помощью последовательных *IF ... THEN ...; ELSE ...*

- либо использованием переходов по массивам меток; действительно, если объявлено

DECLARE TABMET (100) LABEL;

можно дать значение элементам этого массива

TABMET(53) = M;

где M – метка программы, которая участвует, например, в

M : I = I + 1;

после этого можно выполнить оператор

GOTO TABMET(I)

который эквивалентен индексированному переходу в ФОРТРАНе. Точно так же в ПЛ/1 можно оперировать с присваиваниями переменным или элементам массивов типа «метка», как со всякими другими. Этот вид обработки относится к сомнительным средствам и полностью запрещается, если результатом должна быть ясная и надежная программа.

Понятие перехода к элементу таблицы меток соответствует практике программирования в машинных языках (или на языках уровня ассемблера), используемой обычно трансляторами для перевода операторов типа *CASE OF* или индексированного *GOTO;* речь идет о **таблице переходов**.

г) *Понятие таблицы переходов*

Хотя мы будем говорить о технике программирования на языке ассемблирования, познакомиться с понятием таблицы переходов интересно. Пусть существует машина с регистрами, обозначенными *R1, R2, ...* обладающая командой типа

ОПЕРАЦИЯ $r_{рабоч}$ *АДР*($r_{индекс}$)

где *ОПЕРАЦИЯ* указывает код операции, $r_{рабоч}$ и $r_{индекс}$ – номера регистров, *АДР* – адрес, а ($r_{индекс}$) – возможное указание индексации с помощью регистра $r_{индекс}$, т.е. если это указание имеет место, то адрес, используемый в операторе, вычисляется:

(*АДР* + (содержимое регистра $r_{индекс}$ в момент выполнения))

Символические обозначения позволяют нам давать командам метки. Например,

МЕТКА1: ЗАГРУЗИТЬ R2, 257 (R3)

это помеченная команда, которая загружает в регистр *R2* содержимое ячейки с адресом (257 + (содержимое регистра *R3*)). Другой оператор–команда *НА*, где поле $r_{рабоч}$ не участвует:

НА МЕТКА1 (R2)

осуществляет переход к (*n* – 1)–й команде вслед за командой, помеченной меткой *МЕТКА1*; *n* – содержимое регистра *R2* в момент выполнения. Будем допускать, что адреса могут быть символическими именами, указывающими места в памяти, выделенные переменным, как в

МЕТКА: ЗАГРУЗИТЬ 2, IJK(R3)

Предположим также, что каждый оператор занимает в памяти место единичной длины.

При таких условиях оператор АЛГОЛа W

```
CASE I OF
    BEGIN A1; A2; ... AN END
```

может быть представлен следующим образом:

```
        ЗАГРУЗИТЬ R1, I
        «код выдачи сообщения об ошибке или отклонении, если содержимое
        регистра R1 отрицательное, нулевое или превышающее N»
МЕТКА:  НА МЕТКА (R1)
        НА К0Д1
        НА КОД2
        ...                    } (4)
        НА КОД N
К0Д1:    код для A1
        НА ПРОДОЛЖЕНИЕ
К0Д2:    код для A2
        НА ПРОДОЛЖЕНИЕ
        ...
КОДN:   код для AN
ПРОДОЛЖЕНИЕ:    продолжение программы
```

После начальной проверки ошибки индексированный переход

МЕТКА: НА МЕТКА (R1)

выполнит переход к команде ряда (*МЕТКА* + значение *I*), т.е. к команде

НА КОД$_i$ (i = значение I)

которая сама вызовет выполнение кода для A_i с последующим переходом на продолжение программы. «Таблица переходов» – это совокупность команд (4), позволяющих работать с переходами на желаемую метку.

Программа, реализующая индексированный переход, выглядит похоже, в ней будут только опущены команды *НА ПРОДОЛЖЕНИЕ*

III.6. Обсуждение: управляющие структуры и систематическое программирование

Читатель, хорошо знакомый с каким–нибудь языком программирования, нашел бы, вероятно, в высшей степени ограничительными предложенные в этой главе структуры, особенно если он привык пользоваться переходами с помощью оператора

GOTO, произвольно допускаемыми во многих языках. Мы же использовали этот оператор исключительно в «моделировании» некоторых управляющих структур, представленных в качестве фундаментальных, а не как управляющую структуру саму по себе.

На деле же мы не ввели никакого принципиального ограничения. Бём и Якопини доказали в 1966 т. (см. [Бём 66]) следующий результат:

Для всякой программы, выраженной произвольной блок–схемой, существует эквивалентная ей программа (т.е. выполняющая те же преобразования данные → результаты с помощью тех же вычислений), так что:

- операции над переменными те же, что и в исходной программе;
- сохраняются все переменные исходной программы с возможным добавлением некоторого числа **логических переменных** (имеющих два возможных значения):
- единственными используемыми управляющими структурами являются

- цепочка
- цикл пока

Предложенных структур (и даже в действительности только двух из них) достаточно, чтобы описать все алгоритмические задачи. Некоторые другие ограниченные подмножества управляющих структур обладают тем же свойством; можно, например, построить любой алгоритмический процесс с помощью *альтернативы* и *вызова подпрограмм* (возможно, рекурсивного; ср. гл. VI).

Теорема Бёма и Якопини является теоретическим обоснованием для введения наших управляющих структур, но не причиной, по которой рекомендуется их исключительное использование. Таких причин три: средства написания алгоритмов, которые представляют эти структуры, улучшают **читаемость** программ, облегчают **общение** между программистами и делают возможным **доказательство** некоторых свойств программ.

Эти три пункта отвечают целям, на которые ориентирован метод декомпозиции сложных задач, называемый **структурным программированием** [Дал 72]. Улучшение *читаемости* программы позволяет понимать программы при чтении сверху вниз так, что их текст (статическое представление алгоритма) моделирует, насколько возможно, их выполнение (динамическое состояние алгоритма). *Общение* между программистами – один из определяющих элементов в успехе или неудаче программистского проекта. Наконец, более или менее строгое *доказательство* правильности программы, это в конечном итоге единственное средство убедиться в этом самому и убедить других.

Мы будем подробно анализировать эти цели в гл. VII**.** Представим себе на мгновение – мало кто от этого бы отказался, – что желательно их достичь. Задача состоит в том, чтобы это не оставалось благими пожеланиями; необходимость строгой дисциплины, которую требует исключительное применение представленных здесь или сходных структур, не для всех еще неоспоримо убедительна. Фактически с 1968 г., когда Дейкстра опубликовал письмо [Дейкстра 68], поставившее под сомнение неограниченное употребление передач управления в языках программирования, развязалась живая полемика.

Не желая подливать масло в огонь, мы принимаем в этой книге несколько догматическую позицию, ограничившись исключительно описанными выше управляющими структурами. В большинстве случаев это ограничение не вызовет ощутимых затруднений; нужно признать, однако, что при некоторых обстоятельствах (в частности, исключение рекурсии в VI.3) «корсет будет немного узок». Два

затруднительных случая – это циклы «n + 1/2 итераций» (упражнение III.4) и обработка ошибок; добавим, впрочем, что это не означает необходимости всеобщего возвращения к *GOTO,* это значит всего лишь, что ограниченного обобщения представленных здесь структур вполне достаточно (см., например, [Кнут 74] или [Арсак 77]).

Мы решили держаться в строгих рамках цикла, ветвления и цепочки, описанных здесь, потому что неудобства, связанные с исключительным применением этих структур, нам кажутся в значительной мере компенсированными их преимуществами: их простота выражения; их естественный характер (критерий с очевидностью субъективный); их общность и выразительность, которые позволяют с помощью комбинаций базовых элементов выразить все алгоритмические процессы; их небольшое количество и их связь (как показывает случай Даниила Столпника, возможность связать себя с некоторой «структурой» всегда вселяет уверенность. А разве это извращение?); наконец, легкость, с которой они применяются к элементам доказательства правильности программ, проблеме, которая становится непреодолимой при допущении анархии с точками ветвления.

Существуют методы – доказательство Бёма и Якопини это один из них, – сводящие произвольную программу к программе «цепочка + **пока**». Однако это не имеет ничего общего со «структурным программированием», которое представляет собой не технику переделки программ для приведения их в соответствие с эстетическими канонами, а скорее является методом, используемым на всех стадиях программирования от замысла до написания и отладки программ. Таким образом, надо с самого начала пытаться выразить программируемую задачу с помощью примитивных структур, определяющих ясный и простой способ мышления, и стараться сохранить его в ходе всех последовательных этапов разработки программы. В гл. VIII эта мысль уточняется и обобщаются на другие аспекты программирования те методологические принципы, которые в общих чертах намечены здесь в связи с управляющими структурами.

БИБЛИОГРАФИЯ

Вопрос об управляющих структурах был со всей отчетливостью поставлен Э. Дейкстрой в 1968 г. [Дейкстра 68] (см. также статью того же периода [Уилкс 68]). Свои положения Э. Дейкстра развил в [Дейкстра 72], заложив основы «структурного программирования». Более поздняя работа того же автора [Дейкстра 76] представляет и вдохновляет на исключительное использование блока из двух других базовых структур – недетерминированного переключателя и недетерминированного цикла (III.3.2.2).

Исходная статья Дейкстры вызвала бурную реакцию и можно было бы процитировать сотни ссылок; следует откровенно сказать, что немногие из них действительно содержали что–либо новое. С интересом читается [Кнут 74] – большое исследование, предлагающее завуалированный возврат к *GOTO,* а также [Ледгард 75], который выступает против статьи Кнута и дает аргументы в пользу исключительного применения структур, эквивалентных описанным в этой главе.

Аксиоматика управляющих структур (III.4) принадлежит Хоару [Хоар 69]. Последующие статьи были распространены на более общие управляющие структуры, соответственно увеличилась сложность аксиом: [Хоар 71 c], [Хоар 72 b]. В [Дейкстра 76] также приведена интересная аксиоматика для управляющих структур, близких к используемым в этой книге, где проблема завершимости циклов рассматривается в том же формализме, что и другие свойства (инварианты и т.д.), а не по отдельности, как у Хоара. Следует отметить, что это не единственные формальные системы среди тех, что позволяют доказывать свойства программ. Например, полезным было установление [Донегю 76] связи между аксиоматикой Хоара и другой важной теорией – Де Скотта. См. также [Хантлер 76], [Вегнер 72] и [Маркотти 76], в которых сделан обзор ряда

других систем[1].

УПРАЖНЕНИЯ

III.1. Сюрприз мусорной корзины

Приведенный ниже обрывок программы был обнаружен в мусорной корзине одного вычислительного центра. В ней используется фортрановский «ископаемый» оператор, известный под именем «арифметический *IF*» (см. III.5.3.2.б, ФОРТРАН**).**

ФОРТРАН

```
100     CONTINUE
        IF (J – 1) 101, 102, 102
102     CONTINUE
        IF (ITOUR – 1) 105, 105, 106
105     CONTINUE
        NOT1 = NO1
        IOUI = 1
        GOTO 120
106     CONTINUE
        NOT1 = N02
        IOUI = 0
        GOTO 120
101     CONTINUE
        IF (ITOUR – 1) 103, 103, 104
103     CONTINUE
        NOT1 = NO2
        IOUI = 1
        GOTO 120
104     CONTINUE
        NOT1 = NO1
        IOUI = 0
120     CONTINUE
```

а) Что делает эта программа?

б) Можете ли вы написать ее проще?

в) Можете ли вы найти вокруг себя программы, использующие сложные «деревья решений», сложность которых определяется скорее алгоритмом, чем задачей? Как влияют на эту сложность управляющие структуры используемого языка программирования?

III.2. Слияние двух массивов

Приведенная ниже программа взята из учебника введения в программирование на ПЛ/1. Речь идет о решении упражнения, состоящего в построении упорядоченного массива *Z* из элементов массивов *X* и *Y*. *X* и *Y* предварительно упорядочены. (После метки *FIN – КОНЕЦ* – следовали операторы печати массива *Z*.)

[1] Список публикаций по управляющим структурам могут дополнить статьи на русском [Лавров 78] и [Головкин 78]. – *Прим. перев.*

ПЛ/1

```
1            DECLARE (X(50), Y(50), Z(100)) DECIMAL FIXED;
2         /* X ИНДЕКСИРОВАН I, Y – J, Z – K*/
3            I = 1; J = 1 ;
4         DO K = 1 TO 100;
5           IF X(I) < Y(J) THEN DO;
6              Z(K)=X (I) ;
7              I = I + 1;
8         IF I < 51 THEN GOTO FBOUCLE;
9                        ELSE GOTO YDANSZ;
10                END;
11                       ELSE DO;
12             Z(K)=Y(J);
13             J = J = 1;
14             IF J < 51 THEN GOTO FBOUCLE;
15                       ELSE GOTO XDANSZ; END;
16        FBOUCLE: END;
17        /* НИКОГДА НЕЛЬЗЯ ПЕРЕЙТИ ЧЕРЕЗ ЭТОТ
             КОММЕНТАРИЙ */
18        YDANSZ: DO J1 = J TO 50;
19                       K = K + 1;
20                       Z(K)=Y(J1);
21                END;
22        XDANSZ: DO I1 = I TO 50;
23                       K = K + 1;
24                       Z(K) = X(I1);
25                END;
26        FIN: ...
```

а) Что означает первая ветвь альтернативы в строках 8–9? Есть ли более ясное средство для реализации того же эффекта?

Что вы думаете о физическом расположении программы (смещениях)?

Какая польза от *ELSE* в строках 9 и 11?

К какому *THEN* относится *ELSE* из строки 11?

Что означает выражение «перейти через комментарий» (строка 17)?

Означает ли *YDANSZ* (строка 18), что значения *Y* были занесены в *Z* или их туда надо поместить?

б) Можете ли вы выразить этот же алгоритм более ясным способом (на ПЛ/1 или на другом языке)?

Можете ли вы доказать правильность вашей программы?

в) Можете ли вы представить какой–нибудь другой алгоритм?

III.3. Введение в программирование

Эта программа заимствована из учебника введения в программирование на ФОРТРАНе. Речь идет об изображении на печатающем устройстве графика функции, значения которой заданы элементами *Z(1)*, *Z(2)*, ..., *Z(N)* массива *Z*. Интерес представляют только значения функции, содержащейся между *ZMIN* и *ZMAX* (читатели, не знающие хорошо ФОРТРАН, могут игнорировать строки 0 и 19; обозначения, относящиеся к подпрограммам, будут введены только в следующей главе;

то же замечание относится к подробностям написания 16 и 17).

ФОРТРАН

```
0                SUBROUTINE GRAPH (Z, ZMIN, ZMAX, N LA)
1                DIMENSION Z(1000), F(16), G(7)
2                IF (LA) 70, 80, 10
3       80       READ (6,90) (G(J), J = 1,7)
4       90       FORMAT (12A6)
5                DO 500 I = 1,16
6       500      F(I) = G(7)
7                PPAS = (ZMAX – ZMIN)/94.
8                PAS = PPAS * 6.
9                LA = 1
10      70       DO 50 I = 1, N
11               V=(ZMAX – Z(I))*(Z(I) – ZMIN)
12               IF(V) 200, 40, 40
13      40       L = (Z(I) – ZMIN)/PAS
14               M = (Z (I) – ZMIN – FLOAT (L) * PAS)/PPAS
15               F(L + 1) = G(M + 1)
16      200      WRITE (6,60) (F(J), J = 1, 16), I
17      60       FORMAT (X, 16A6, 16)
18      50       F(L + 1) = G(7)
19               RETURN
20               END
```

Требуется несколько пояснений. *G* – это массив, предназначенный для хранения семи текстов, каждый из которых содержит 6 литер. Тексты образуются из пробелов и нуля или одной звездочки: **bbbbbb*; *b*bbbb*; *bb*bbb*; *bbb*bb*; *bbbb*b*; *bbbbb** и *bbbbbb* (последний текст, являющийся значением *G(7)*, используется в строке 6).

Рассматриваемый пример представляет собой подпрограмму, в которой LA – один из ее параметров; строки 2–4 обеспечивают, что первый вызов подпрограммы (при условии что LA вначале равно нулю) будет порождать чтение семи значений из *G* (ни какое–либо упоминание этого факта, ни объяснение значения *LA* не сопровождают текст программы[1]).

Что вы думаете о методе, который состоит в том, что значения семи констант вводятся с помощью специальной подпрограммы чтения?

Как пользователь этой подпрограммы узнает, где ему разместить свои константы в потоке данных?

Константа *94* (строка 7) – это число свободных колонок в строке печатающего устройства, использованного авторами упомянутого учебника. Такое объяснение, напротив, не годится для значения константы 6 (строка 8). Сможете ли вы объяснить это значение? (*Рекомендация:* не тратьте много времени на этот вопрос.) Считаете ли вы целесообразным включать таким образом константы в «вычислительную» часть общей программы?

Формат 90 строки 4 означает, что читаются 12 текстов по 6 литер в каждом; действительно, машина CDC 6600 (на которой авторы учебника проверяли свою программу) может разместить 6 литер в вещественном числе (напомним, что

[1] Конкретное применение этого примера в учебнике использует константу 0 в качестве фактического параметра, соответствующего *LA* (!).

ФОРТРАН не имеет переменных типа ЛИТЕРА или СТРОКА). Считаете ли вы, что именно так следовало воспользоваться возможностями машины?

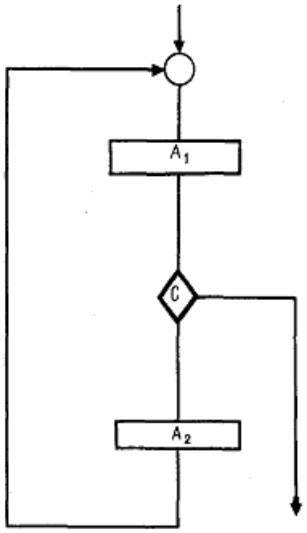
F – массив, содержащий изображение строки (94 литеры). Заметьте, что при 6 литерах на вещественное *F* с границами от 1 до 16 содержит 6 x 16 = 96 литер, а не 94. Строки 13 и 15 устанавливают шестилитерный текст из *G* в соответствующий элемент *F*. Понятны ли вам подробности? Можете ли вы доказать, что индексы в строке 15 законны (т.е. что $0 \le L \le 15$ и $0 \le M \le 6$)?

Что вы можете сказать еще об этой программе?

Напишите простую программу, выполняющую сформулированную задачу (*рекомендация:* массив *G*, переменные *LA*, *PPAS* и *PAS* лишние. Использовать «строковые константы» ФОРТРАНа).

III.4. N с половиной итераций

Задача, которую часто вспоминают сторонники оператора *GOTO*, соответствует изображенной схеме программы (цикл, выполняемый «n +½ раз»).



а) Можете ли вы дать конкретный пример, отвечающий этой схеме?

б) Можете ли вы дать программу, решающую этот тип задач, используя только управляющие структуры, введенные в этой главе. Сравните эту программу с программой, использующей *GOTO* (простота, элегантность, ясность...)?

в) Можете ли вы придумать новую управляющую структуру, решающую эту задачу, с простыми обозначениями и сформулировать ее свойства таким же образом, как в III.4?

III.5. Последовательный поиск

Другой пример, часто используемый в поддержку *GOTO*, – линейный поиск в таблице (конец разд. III.4). Сравните (ясность, эффективность...) следующую программу с программой разд. III.5:

```
{поиск x среди элементов T[m:n]}
для i от m до n повторять
    | если T [i] = x то
    |     | индекс ← i {сохранение i, зависит от языка}
    |     | на найден;
не найден: ... {обработка ситуации, в которой никакой элемент
                T не равен x} ...;
           на продолжение;
найден: ... {обработка ситуации, где T [индекс] = x} ...
...
продолжение: ... {продолжение программы}
```

В какой версии легче всего соединить точные утверждения с некоторыми точками программы и доказать их?

Предложена также следующая программа, в которой искусственно добавляется x в конце T, чтобы быть уверенным, что программа завершается обнаружением x:

```
{поиск x среди элементов T[m:n]}
i ← m; T[n + 1] ← x;
пока T[i] ≠ x повторять
      | i ← i + 1
{в этой точке: или i ≤ n и T [i] = x,
                или i > n и ни один
                    элемент из T [m : n] не равен x}
```

Сравните эту программу с предыдущими (ясность, эффективность ...). Легко ли на практике присоединить дополнительный элемент к массиву? Стоит ли это делать?

III.6. Кено и три маленькие шустрые горошины (диалог)

Раймон Кено предложил под названием «Сказка на ваш вкус» текст, приведенный на Рис. III.4 (отрывок из *Улипо,* изд–во «Галлимар», 1972).

Требуется написать программу, «рассказывающую» эту сказку по желанию «читателя», сидящего перед диалоговым терминалом. Выполнение программы состоит из таких этапов: программа печатает несколько строк, за которыми следует вопрос (типа: *«Предпочитаете ли вы, чтобы они спали?»*). Читатель отвечает, набирая на клавиатуре «Да» или «Нет». Тогда программа выбирает продолжение текста в зависимости от этого ответа. (Предусмотреть, кроме «Да» и «Нет», еще некоторый код, означающий «Достаточно».)

СКАЗКА НА ВАШ ВКУС

Этот текст был представлен на 83–м рабочем совещании Футуристического литературного ателье, которое рассматривает программы для вычислительных машин и для программированного обучения. Эта структура аналогична «древесной» литературе, предложенной Ф.Ле Лионнэ на 79–м совещании.

1. Хотите ли вы узнать историю трех маленьких шустрых горошин?

Если да, перейдите к 4,

если нет, перейдите к 2.

2. Может быть, вы предпочитаете историю трех длинных жердей?

Если да, перейдите к 16,

если нет, перейдите к 3.

3. Может быть, вы предпочитаете историю о трех простых скромных кустиках?

Если да, перейдите к 17, если нет, перейдите к 21.

4. Жили–были когда–то три маленькие горошины, одетые во все зеленое. Они мирно спали в своем стручке. Личики у них были совсем кругленькие, а маленькие носики тихо и ровно посапывали.

Если вы предпочитаете другое описание, перейдите к 9, если вас все устраивает, перейдите к 5.

5. Они не видели снов. Эти маленькие существа вообще никогда не видят снов.

Если вы предпочитаете, чтобы они видели сны, перейдите к 6; иначе перейдите к 7.

6. Они видели сны. Эти маленькие существа все время видят сны, и ночи скрывают их чудесные сновидения.

Если вы хотите узнать эти сны, перейдите к 11, если для вас это безразлично, перейдите к 7.

7. Их миленькие ножки были укутаны в теплые носки, а с рук они никогда не снимали черные шерстяные перчатки.

Если вы предпочитаете перчатки другого цвета, перейдите к 8, если этот цвет вас удовлетворяет, перейдите к 10.

8. Они никогда не снимали голубые шерстяные перчатки. Если вы предпочитаете перчатки другого цвета, перейдите к 7,

если этот цвет вам подходит, перейдите к 10.

9. Жили–были три маленькие горошины, которые обошли весь свет, бродя по дорогам. К вечеру, утомленные и усталые, они очень быстро уснули.

Если вы хотите знать, что было дальше, перейдите к 5, иначе перейдите к 21.

10. Всем трем снился одинаковый сон, они в самом деле нежно любили друг друга; словно отраженные в трех зеркалах, они видели похожие сны.

Если вы хотите узнать их сон, перейдите к 11,

иначе перейдите к 12.

11. Им снилось, что они уселись ужинать в харчевне, и, открыв крышку кастрюли, они увидели, что это был суп из чечевицы. От ужаса они проснулись.

Если вы хотите знать, почему они проснулись в ужасе, поищите в энциклопедии слово «чечевица», и не будем об этом больше говорить,

если вы считаете бесполезным углублять этот вопрос, перейдите к 12.

12. Ой–ой–ой! – вскрикнули они, открыв глаза. Ой–ой–ой! что за сон мы увидели. Это не к добру, – сказала первая горошина. Да, – сказала вторая, – это так, я боюсь. Не тревожьтесь, – сказала третья, которая была самая умная, – надо не нервничать, а разобраться. Я сейчас попробую вам все объяснить.

Если вы хотите сразу же узнать толкование этого сна, перейдите к 15, если, напротив, вы хотите знать, как на это прореагировали две другие горошины, перейдите к 13.

13. Не заговаривай нам зубы, – сказала первая. – С каких это пор ты научилась толковать сны? Да, с каких пор? – прибавила вторая.

Если вы тоже хотите знать, с каких пор, перейдите к 14, иначе перейдите все–таки тоже к 14, потому что вы ничего не узнаете больше.

14. С каких пор? – вскричала третья. Разве я знаю? Да, я умею. Сейчас увидите!

Если вы хотите увидеть, перейдите к 15,

если нет, то тоже перейдите к 15, но вы ничего не увидите.

15. Хорошо, посмотрим! – сказали сестры. Мне не нравятся ваши насмешки, – ответила тогда третья горошина. – И вы ничего не узнаете. Кстати, пока мы здесь все довольно живо обсуждаем, не уменьшились ли ваши страхи? Или, может, совсем рассеялись? Тогда стоит ли копаться в глубинах вашего подсознания мотыльковых? Пойдемте скорее к фонтану, умоемся и порадуемся этому веселому утру в чистоте и добром здравии! Сказано – сделано: они вылезли из своего стручка, спустились осторожно на землю и затем быстро и весело добрались до фонтана.

Если вы хотите знать, что произошло у фонтана, перейдите к 16, если вы этого не хотите, перейдите к 21.

16. Три большие длинные жерди смотрели, что они делают.

Если три большие длинные жерди вам не нравятся, перейдите к 21, если они вас устраивают, перейдите к 18.

17. Три простых скромных кустика смотрели, что они делают.

Если вам не нравятся три простых скромных кустика, перейдите к 21, если они вам подходят, перейдите к 18.

18. Видя, что они так на них глазеют, три шустрые маленькие горошины смущенно отвернулись.

Если вы хотите узнать, что они потом сделали, перейдите к 19, если вы этого не желаете, перейдите к 21.

19. Они быстренько побежали к своему стручку, укрылись в нем и снова заснули.

Если вы хотите знать продолжение, перейдите к 20, если вы этого не желаете, перейдите к 21.

20 А продолжения–то и нет, потому что сказка кончилась.

21 И в этом случае сказка тоже кончилась.

Раймон Кено

Рис. III.4 Текст–игра Раймона Кено.

III.7. Только программное управление

Рассмотрите снова программу из разд. III.3.6 (лексический мини–анализатор), осуществив управление только с помощью программы, т.е. исключив всякие обращения к массиву Переход. Обсудите новую версию по отношению к описанной в книге.

III.8. А для строк?

Возможно ли изменить диаграмму перехода рис. III.2 (или, что, по существу, является тем же самым, массив Переход) так, чтобы можно было анализировать строковые константы, которые существуют в многочисленных версиях ФОРТРАНа, т.е.

'a b ... *l*'

где a, b, ..., *l* – произвольные литеры с тем лишь соглашением, что литера кавычки удваивается, если она должна присутствовать внутри строковой константы? Можно ли изменить диаграмму таким образом, чтобы можно было анализировать «холлеритовские константы» стандартного ФОРТРАНа

nHab...l

n литер

Те же вопросы ставятся и в том случае, когда используется не программа, управляемая таблицей переходов, а программа задачи III.7.

III.9. Индийское возведение в степень

[Дейкстра 72] Строго доказать с помощью аксиоматики Хоара (III.4), что следующая программа вычисляет $z = A^B$, где A и B –целые константы; предполагается, что $A > 0$ и $B > 0$.

```
переменные x, y, z: ЦЕЛЫЕ;
x ← A; y ← B; z ← 1;
пока y ≠ 0 повторять
    если y нечетное то
        z ← z * x
        y ←y – 1
    иначе {y – четное}
        x ← x × x;
        y ← y/2
```

рекомендация: докажите, что свойство

$\{x > 0, y \geq 0$ **и** $z \times x^y = A^B\}$

является инвариантом цикла.

Каковы преимущества этого алгоритма по сравнению с тривиальным методом вычисления A^B?

III.10. Расширения аксиоматики

Строго определите таким же образом, как базовые структуры, исследованные в III.4, следующие структуры:

а) **повторять** A **до** C

б) **для** i **от** m **до** n **шаг** p **повторять** A

в)
```
выбрать
    c1:a1,
    c2:a2,
    ...
    ...
    cn:an
иначе
    a0
```

г) *CASE OF* (АЛГОЛ W)

III.11. Неразрешимость

Показать (ср. III.4.5), что нельзя написать программу P, способную по заданному тексту p некоторой программы и набору d ее данных определить, завершается или нет выполнение программы p с данными d (*рекомендация:* построить, исходя из P, программу, которая могла бы быть применена к своему собственному тексту, и вывести из этого противоречие).

ГЛАВА IV.
ПОДПРОГРАММЫ

ПОДПРОГРАММЫ

Подпрограмма – одно из фундаментальных средств, имеющихся в распоряжении программиста; это средство абстракции и искусственного расширения возможностей языка и машины, имеющейся в распоряжении. Эти аспекты были затронуты в гл. III (где подпрограмма рассматривалась как одна из управляющих структур): к ним мы вернемся в гл. VIII при обсуждении роли декомпозиции на подпрограммы в современной методологий программирования. Настоящая глава описывает технические аспекты понятия подпрограммы; речь идет о важных концепциях, связанных с проблемой общения: общение между программными модулями, между «квазипараллельными» программами, передача и обобществление данных.

IV.1. Введение

В гл. III мы видели, как подпрограммы снабжают программиста удобным средством *абстракции,* позволяя сжато именовать достаточно сложные последовательности действий.

Настоящая глава посвящена техническим аспектам понятия подпрограммы, которые приобретают огромное значение, как только большие размеры задачи, которую надо решить, приводят к появлению многочисленных «программных модулей», к которым в той или иной степени применяются строгие принципы «модульного» программирования. После знакомства с проблемой обозначений, относящихся к подпрограммам (IV.2), и с ее решениями, принятыми ФОРТРАНОМ, АЛГОЛом W и ПЛ/1 (IV.3), мы обсудим зачастую плохо усваиваемый вопрос об обмене информацией между программными модулями (IV.4 и IV.5), рассмотрим управление данными в подпрограммах (IV.6) и коснемся в IV.7 некоторых подходов к расширению понятия подпрограммы, в частности *сопрограммы.* Одна из проблем, связанная с подпрограммами, это *рекурсия;* ее важность оправдывает то, что ей посвящается отдельная глава VI.

IV.2. Определения и проблемы обозначений

IV.2.1. Определения

В гл. III подпрограмма определялась как средство *именования* сложного *действия.*

Здесь мы обобщим это определение, рассматривая подпрограмму как совокупность операторов, вычисляющих некоторое число результатов, зависящих от некоторого числа аргументов. Аргументы и результаты подпрограммы вместе называются ее параметрами; иногда слово параметр употребляют как синоним аргумента[1]).

Таким образом, подпрограмма определяется как функциональное отношение между возможными аргументами и соответствующими результатами (Рис. IV.1).

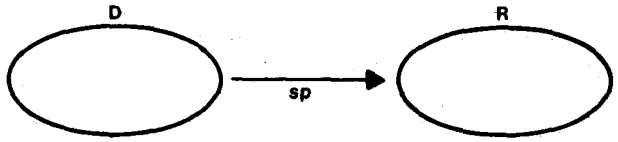


Рис. IV.1 Функциональное определение подпрограммы.

Точно так же была определена в гл. III *программа* вообще. Главное различие состоит в том, что подпрограмма может действовать в интересах другой программы или подпрограммы. Далее всюду, когда это не будет приводить к путанице, мы в равной мере будем употреблять термины *программа, подпрограмма,* а также *программный модуль*. О программе, которая обращается к другой для выполнения некоторого действия, мы будем говорить, что она *вызывает* другую программу с помощью оператора, называемого **вызовом подпрограммы**; будем различать **программу вызывающую** и **программу вызываемую**. Когда выполнение вызываемой программы прекращается, говорят, что она осуществляет возврат к вызывающей программе, которая продолжает свое выполнение с оператора, следующего (в цепочке) за вызовом.

Программы, которые могут вызываться другими программами, могут иметь дело не только с аргументами, известными в момент вызова, и результатами, выдаваемыми в момент возврата, но также при некоторых условиях с величинами, принадлежащими вызывающим программам; такая обработка может определенным образом преобразовывать эти величины. Поэтому мы будем выделять в числе параметров, кроме **аргументов** и **результатов**, те, которые будут называться **модифицируемыми параметрами.** Важно отметить, что эта категория параметров не является концептуально необходимой: можно считать, что подпрограмма, модифицирующая параметр d, практически только вырабатывает результат d_2 в зависимости от параметра d_1; равного d. Однако понятие модифицируемого параметра очень удобно на практике.

Наконец, любая программа может использовать, кроме аргументов и результатов, еще и *промежуточные переменные,* необходимые на этапах вычислений (III.2.1). В случае подпрограммы говорят скорее о **локальных переменных,** чтобы подчеркнуть, что эти объекты не доступны за пределами подпрограммы, которая их использует. Проблемы, связанные с локальными переменными, будут изучены в IV.6.

IV.2.2. Определение и вызов; формальные параметры, фактические параметры

Для подпрограммы важно различать понятие ее *определения* и понятие ее *вызова.*

Чтобы лучше различать эти два термина, предлагается аналогия из математики. Когда пишут

«существует функция $f(x, y) = x^9,+ 8x^3y^7+ xy^2$», тем самым *определяют* объект, функцию f, с помощью двух «связанных» переменных x и y; как подсказывает

[1] В некоторых работах, в частности по ПЛ/1, называют соответственно «аргументом» и «параметром» то, что мы ниже называем «фактическим параметром» и «формальным параметром» (или «фактическим аргументом» и «формальным аргументом»).

выражение «связанная переменная», имена x и y не имеют никакого собственного смысла, и они могли бы таким же точно образом определить функцию, описанную как

«существует функция $f(u, v) = u^9 + 8u^3v^7 + uv^2$»

или даже

«существует функция $f(y, x) = y^9 + 8y^3x^7 + yx^2$». Определив функцию f, ее можно *использовать* (в программировании говорят *вызывать*), задавая ей «фактические» значения, которые будут вставать на место связанных переменных в вычислении f.

Каждая из трех следующих записей однозначно определяет значение

а) «f (3,5)» (равное $3^9 + 8 \times 3^3 \times 5^7 + 3 \times 5^2$)

б) «f (2, f(sin(π/7), f (f(3/8, 5), f(1, 7)))»

в) «$f(x_1, x_2)$, где x_1 и x_2 – корни уравнения $x^2 + x - 1 = 0$»

Строгое определение этих понятий заставляет обратиться к теории «лямбда–исчисления» (введение в теорию можно найти, например, в [Бердж 75]). Того, что сказано, однако, достаточно для пояснения понятий определения и вызова в том виде, в котором они используются в практике программирования.

Определение, или **декларация,** подпрограммы содержит указание ее имени (которое должно позволять другим программам обращаться к ней), указание числа параметров, которые должны быть ей конкретизированы при вызове (аргументы, результаты, модифицируемые параметры), и типов каждого из них, а также описание действия, которое использует эти параметры. Такое описание имеет вид программы, где аргументы представлены некоторыми именами, называемыми **формальными параметрами,** которые играют роль «связанных переменных» в математике. Синтаксически определение подпрограммы является ее объявлением: так же, как объявление переменной, массива и т.д., определение подпрограммы не предписывает выполнения какого–либо действия (хотя и содержит операторы), а просто ставит в соответствие имя (типа «программа») некоторому объекту (элементу программы).

При таком определении подпрограммы всякая программа может содержать **вызов** этой подпрограммы, задаваемый ее именем, за которым следует список объектов, определяемых в вызывающей программе (константы, переменные, массивы и т.д.); некоторые из этих объектов – **аргументы**, значения которых передаются вызываемой программе, другие – **результаты**, которым вызываемая программа присваивает значения, наконец, третьи – модифицируемые **параметры.** Эти передаваемые подпрограмме объекты называются **фактическими параметрами,** каждый из которых должен соответствовать по типу одному из формальных аргументов, фигурирующих в определении подпрограммы.

Пример:

Подпрограмма с именем макстаб100 может вычислять самый большой элемент в массиве из 100 целых и индекс (или один из индексов) такого элемента. Эта подпрограмма имеет своими формальными параметрами:

- аргумент, массив, который можно обозначить таб;
- два результата, которые можно назвать максимум и индекс–макс.

Если есть определение этой подпрограммы, то программа P, начало которой имеет вид

```
массив параб [1 : 100] : ЦЕЛЫЙ;
переменные x,y : ЦЕЛЫЕ;
для i от 1 до 100 повторять
    | параб [i] ← 50 – (15–i)^2
```

может содержать вызов макстаб100 с фактическими параметрами параб, x и y, которые соответствуют формальным параметрам таб, максимум и индекс–макс. После этого

вызова параб останется неизменным, x станет равным 50, а y будет равно 15.

IV.2.3. Проблемы обозначений; подпрограммы «операторы» и «выражения»

Определение подпрограммы

Для определения подпрограммы воспользуемся принятыми в обычных языках программирования понятиями заголовок и тело подпрограммы. Так, для предыдущего примера подпрограмма макстаб100 определяется следующим образом:

```
программа макстаб100 (аргумент массив таб[1 : 100] : ЦЕЛЫЙ;
                      результаты: максимум, индекс–макс : ЦЕЛЫЕ)
    максимум ← ∞
    для i от 1 до 100 повторять
        если таб [i] > максимум то
            максимум ← таб [i];
            индекс–макс ← i
```

В этом примере «заголовок» образован двумя первыми строками, которые задают имя подпрограммы и объявляют способом, сходным с объявлениями переменных, параметры подпрограммы типа аргумент и результат (здесь отсутствуют модифицируемые параметры).

«Тело» подпрограммы образовано операторами, которые следуют за «заголовком»; в этом примере они вычисляют максимум и индекс–макс в зависимости от аргумента таб.

Вызов подпрограммы

Чтобы вызвать подпрограмму, достаточно написать в вызывающей программе имя этой подпрограммы вместе с заключенным в скобки списком фактических параметров, каждому из которых предстоит заменить соответствующий формальный параметр в списке формальных параметров, участвующем в определении подпрограммы (точные свойства этой замены изучаются в IV.4).

Так, упомянутая выше вызывающая программа P может содержать вызов

макстаб100 (параб, x, y)

Этот вызов есть оператор; его выполнение эквивалентно (эта эквивалентность будет уточнена в IV.4) выполнению тела подпрограммы после соответствующей замены параметров

```
x ← –∞;
для i от 1 до 100 повторять
    если параб [i] > x то
        x ← параб[i];
        y ← i
```

Подпрограммы–«выражения»

Может оказаться полезным обобщение введенного выше понятия, чтобы позволить подпрограмме задавать не только действие (оператор), но также и значение (выражение). Рассмотрим программу, вычисляющую (приближенно) квадратный корень из любого вещественного числа. В нашем предыдущем формализме эта подпрограмма запишется в виде

```
программа квадр–корень (аргумент x : ВЕЩ; результат : ВЕЩ)
    ...
    {вычисление y = √x }
```

Программа, использующая квадратный корень из 27,365, например, должна будет вычислять его с помощью оператора

квадр–корень (27.365, а)

где а – объявленная ВЕЩЕСТВЕННАЯ переменная. Это, очевидно, достаточно неудобно: в математике привычно рассматривать

корень (27.365)

как значение, используемое в выражении. Чтобы разрешить такое описание, мы добавим к подпрограммам типа «оператор», рассмотренным выше, так называемые подпрограммы типа «выражение», где имя подпрограммы играет роль параметра результат. Заголовок такой подпрограммы может обозначаться, например, как

программа корень : ВЕЩ **(аргумент** x **:** ВЕЩ**)**

В теле подпрограммы ее имя рассматривается как параметр результат, а его конечное значение есть значение, «выдаваемое» подпрограммой, или, другими словами, «результат» вызова. Вспоминая, что для x > 0 последовательность $x_1 = x,\ x_n \frac{1}{2}\left(\frac{x}{x_{n-1}} + x_{n-1}\right)$ сходится к $\sqrt{x}$, получим, например, тело следующей подпрограммы, вычисляющей приближение к $\sqrt{x}$ (ε – положительная константа):

```
| корень ← x;
| пока |корень^2 – x| > ε повторять
|     | корень ← 1/2 (x/корень + корень)
```

Тогда можно использовать корень(е) где е – выражение типа ВЕЩ в качестве выражения

```
переменные x, y, z : ВЕЩ;
x ← 7.2; y ← 5.6;
z ← корень(x) – корень (корень(y) + корень(x + y))
{и т.д.}
```

Разумеется, подпрограмма–«выражение» может, кроме своего «результата», иметь произвольное число параметров типа «аргумент», «результат» и «модифицируемый параметр».

Это допущение (похожее на обозначения ФОРТРАНа, ПАСКАЛя и др. языков) не вполне удовлетворительно: с одной стороны, имеет место некоторая потеря симметрии, потому что было бы нормально, если бы тело подпрограммы–«выражения» было скорее выражением, чем оператором; с другой стороны, подпрограмма имеет смысл только в том случае, если при всяком выполнении результат гарантированно получает значение. Однако трансляторы редко проверяют истинность этого условия, такая проверка достаточно трудна, если она возлагается на программиста, это увеличивает вероятность ошибки.

Другое решение, принятое в АЛГОЛе W, состоит в обобщении понятия выражения, позволяющем включать сложные вычисления так, что тело подпрограммы–«выражения» действительно является выражением.

Так, в АЛГОЛе W определены «условные выражения» (II.4.3.4.е), значения которых зависят от условия, и «выражения–блоки», значения которых порождают выполнение некоторого числа операторов. Например, в АЛГОЛе W описание

```
АЛГОЛ W
        BEGIN
        INTEGER X;
        READON (X);
        WHILE X >= 0 DO
              READON (X);
        ABS X
        END
```

означает *выражение,* значение которого есть значение выражения, предшествующего *END,* т.е. *ABS X,* вычисленное после того, как выполнены операторы, следующие за *BEGIN*. Это выражение имеет, таким образом, своим значением абсолютную величину первого прочитанного отрицательного числа, если такое существует. Оно могло бы быть присвоено целой переменной или служить телом подпрограммы–«выражения».

В таких языках, как АЛГОЛ 68 или БЛИСС, понятие «обобщенного» выражения было систематизировано: эти языки обладают единственным понятием («предложение» в АЛГОЛе 68), объединяющим то, что в других языках называется «выражением» или «оператором»: всякое «предложение» может иметь сразу и значение, и эффект действия; так, присваивание

x ← e

имеет в качестве значения величину выражения e, а в качестве эффекта действия – присвоение этого самого значения переменной x. Некоторые «предложения» не имеют значений и рассматриваются поэтому имеющими специальное «пустое» значение: они соответствуют операторам в обычном смысле. Другие предложения, соответствующие обычным «выражениям», не обладают действиями. Принятие такого соглашения дает определению языка замечательную элегантность; однако тому, кто был воспитан в духе разграничения «действия» и «значения», бывает порой трудно понять программу, свободно смешивающую эти два понятия. В АЛГОЛе W, который не ушел далеко, хотя он и обладает выражениями–блоками, условными выражениями и *CASE* (IV.3.2), проблем практически не возникает, потому что выражения–блоки редко употребляются иначе, чем в качестве подпрограмм–«выражений»[1].

IV.2.4. Тип подпрограммы

Функционально определяемые подпрограммы характеризуются типами их аргументов и типами результатов. Совокупность этих типов определяет *тип подпрограммы.* Так, подпрограмма

```
программа p (аргументы x : ЦЕЛ, y : ВЕЩ, z : СТРОКА;
             результаты t : ВЕЩ, и : ЦЕЛ)
                 | ... {любой алгоритм} ...
```

ставит одно ВЕЩ и одно ЦЕЛ в соответствие всякому триплету [ЦЕЛ, ВЕЩ, СТРОКА]. Будем говорить, что эта программа имеет тип

(ЦЕЛ × ВЕЩ × СТРОКА → ВЕЩ × ЦЕЛ)

Подпрограмму с «модифицируемыми параметрами» можно легко привести к этой же схеме: считается, что каждый такой параметр состоит из одного аргумента и одного результата. Если подпрограмма не имеет ни аргументов, ни результатов, можно легко уйти от трудностей, заменяя недостающие типы специальным типом ПУСТО.

Это определение типа программы имеет больше практического смысла, чем это может показаться на первый взгляд: в самом деле, оно позволяет определить тип с точностью до подпрограмм, имеющих своими параметрами программы. Пусть, например, программа, называемая интеграл, такова, что интеграл(f, a, b), где где a и b –

[1] Практически программисты иногда сталкиваются с трудностями, когда «выражение» законно там же, где разрешен и «оператор». В языках алголовского типа знак операции присваивания часто изображается :=, а знак оператора отношения – символом = . Если знак : опущен программистом в присваивании, то «оператор» может быть воспринят транслятором «выражение» типа ЛОГИЧЕСКОЕ: ошибка останется незамеченной (и программа будет неверной) или будет иметь место трудно понимаемое диагностическое сообщение.

вещественные числа, а f – функция, имеет своим значением $\int_b^a f(x)dx$. Функция f могла бы быть любой интегрируемой функцией, например синусом, косинусом, показательной функцией и т.д. Благодаря нашему определению типа подпрограммы можно задать типы параметрам интеграла, как параметрам любой другой программы:

программа интеграл : ВЕЩ (**аргументы** f : (ВЕЩ → ВЕЩ), a, b : ВЕЩ)
| {алгоритм вычисления интеграла от f
| в пределах от a до b, например,
| методом трапеций}...

интеграл

((ВЕЩ → ВЕЩ) × ВЕЩ × ВЕЩ → ВЕЩ)

IV.3. Введение в использование подпрограмм в ФОРТРАНе, АЛГОЛе W, ПЛ/1

В этом разделе описывается базовая структура подпрограмм в наших трех языках, в этом смысле весьма сходных. В следующих разделах мы вернемся к обсуждению более тонких проблем общения между программными модулями и управления памятью.

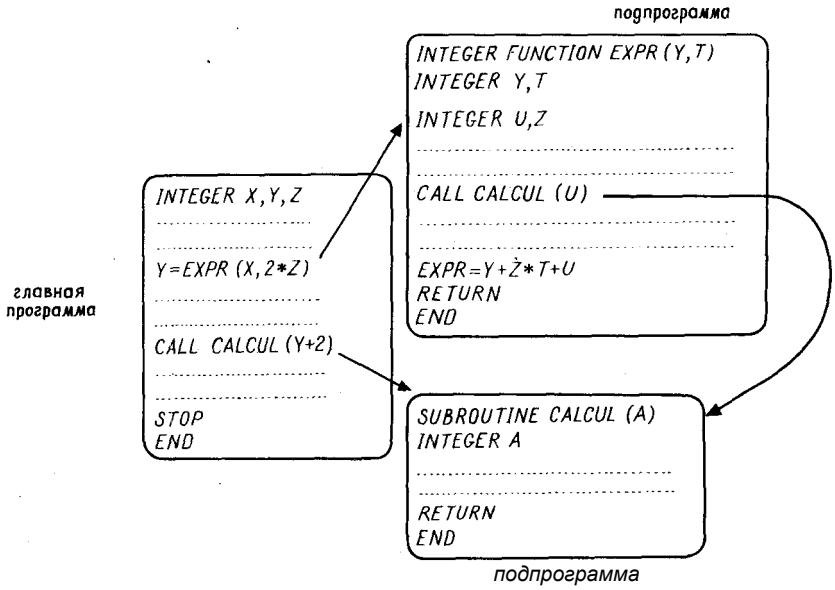


Рис. IV.2 Главная программа и подпрограммы в ФОРТРАНе.

IV.3.1. Подпрограммы в ФОРТРАНе

ФОРТРАН не имеет блочной структуры, и программа в этом языке составляется множеством физически разделенных программных модулей (Рис. IV.2); один из них – **главная программа,** другие являются подпрограммами. В ФОРТРАНе предусмотрено все, чтобы сделать возможной трансляцию одного программного модуля независимо от других.

IV.3.1.1. Подпрограммы–«выражения»

Форма подпрограммы–«выражения», или функции в терминологии ФОРТРАНа, иллюстрируется приведенным ниже примером, в котором находится наименьшее из двух целых[1]:

[1] Минимум двух целых обычно вычисляется в ФОРТРАНе с помощью *MIN0(I, J),* а максимум – с помощью

ФОРТРАН

```
      INTEGER FUNCTION MINIM (X, Y)
      INTEGER X, Y
C     ВЫЧИСЛЕНИЕ МИНИМУМА ИЗ X И Y
      MINIM = X
      IF (Y.LT. X) MINIM = Y
      RETURN
      END
```

В общем виде заголовок подпрограммы–«выражения» записывается в виде

тип FUNCTION имя–подпрограммы (пар1,..., пар n)

где тип – это тип ФОРТРАНа; *имя–подпрограммы* есть идентификатор, означающий результат; *пар1,..., парn* – идентификаторы, означающие формальные параметры; обычно их тип объявляется в подпрограмме (если только не использованы объявления по умолчанию, определяемые первой буквой). Заметьте, что здесь не уточняется, является ли параметр аргументом, результатом или модифицируемым параметром.

Тело подпрограммы–«выражения» – это последовательность операторов, заканчивающаяся директивой *END*. Любое выполнение подпрограммы должно приводить к исполнению оператора *RETURN*, которое возвращает управление вызывающей программе. Перед выполнением оператора *RETURN* переменная *имя–подпрограммы* (в нашем примере *MINIM*) должна получить значение, например присваиванием; именно это конечное значение выдается подпрограммой в вызывающую программу.

Чтобы *использовать* подпрограмму–«выражение» в программном модуле, достаточно написать *имя–подпрограммы,* сопровождаемое списком фактических параметров в скобках.

Далее форма фактических параметров будет уточнена; скажем просто, что речь должна идти о константах, переменных или выражениях, тип которых соответствует типам *формальных* параметров из заголовка подпрограммы. Так, следующая программа использует несколько раз подпрограмму *MINIM*:

```
      INTEGER MINI, L, AT
      ...
      L= MINIM(L, AT)
C     ПРЕДЫДУЩИЙ ОПЕРАТОР ЭКВИВАЛЕНТЕН IF (AT.LT.L) L= AT
      ...
      MIN1 = MINIM (L + 2* AT + 1, MIN1 – L) + AT*7
      ...
      IF (MINIM(L, AT+ 1) .GE. 3 * MINIM(MIN1, 8)) L = 0
      END
```

Как показывает этот пример, обращение типа

MINIM (факт–параметр–1, факт–параметр–2)

играет ту же синтаксическую роль, что и целое выражение. Его значение есть результат вычисления, описанного телом подпрограммы, где формальные параметры заменены фактическими параметрами.

IV.3.1.2. Подпрограммы–«операторы»

Форма подпрограммы–«оператора» в ФОРТРАНе проиллюстрирована

MAX0(I, J); MIN0 и *MAX0* – это «встроенные» функции. Подпрограмма *MINIM* и другие сходные программы этой главы фигурируют здесь только в качестве примеров.

следующим примером:

```
ФОРТРАН
      SUBROUTINE IMPCAR (C, N)
      INTEGER C, N
C     НАПЕЧАТАТЬ СТРОКУ ИЗ 120 ЛИТЕР,
C     ГДЕ N–Я ЛИТЕРА ЕСТЬ C, А ДРУГИЕ – ПРОБЕЛЫ
C     ФОРТРАН НЕ ИМЕЕТ ТИП «ЛИТЕРА».
C     НАИБОЛЕЕ НАДЕЖНЫЙ МЕТОД СОСТОИТ В РАЗМЕ–
C     ЩЕНИИ ЛИТЕРЫ В ЦЕЛОМ (ЗДЕСЬ C,
C     ПРОБЕЛ BLANC, ЗВЕЗДОЧКА–ETOILE, ЛИТЕРА – CARAC).

C     В СЛУЧАЕ НЕПРАВИЛЬНОЙ СПЕЦИФИКАЦИИ
C     ПЕЧАТАЕТСЯ СТРОКА ЗВЕЗДОЧЕК
      INTEGER LIGNE (120)
      INTEGER BLANC, ETOILE, CARAC
      DATA BLANC /' '/, ETOILE /'*'/
C     —— СТАНДАРТ ФОРТРАНА ПРЕДПОЧИТАЕТ 1H И 1H*,
C      А НЕ ' ' И ' * ' ——
      LOGICAL ERREUR
C     ERREUR–ОШИБКА
C     —— ИЗГОТОВЛЕНИЕ СТРОКИ ——
      ERREUR = (N.LT.1).OR.(N.GT.120)
      CARAC = BLANC
      IF (ERREUR) CARAC = ETOLLE
      DO 100 1 = 1, 120
          LIGNE(I) = CARAC
100       CONTINUE
      IF(.NOT.ERREUR) LIGNE(N) = C
C
C      —— ПЕЧАТЬ СТРОКИ ——
      PRINT(10000) LIGNE
10000 FORMAT (1X, 120A1)
C
      RETURN
      END
```

Подпрограмма–«оператор» имеет заголовок вида

SUBROUTINE имя–подпрограммы(пар1, ..., парn)

В отличие от случая подпрограмм–«выражений» список параметров здесь может отсутствовать. Это может оказаться полезным в случае, когда выполняются фиксированные операции, например

```
ФОРТРАН
      SUBROUTINE PAGE
C     PAGE–СТРАНИЦА
C     ПРОДВИНУТЬ БУМАГУ К НАЧАЛУ СЛЕДУЮЩЕЙ
C     СТРАНИЦЫ
      PRINT (10000)
10000 FORMAT (1H1)
      RETURN
      END
```

Тело подпрограммы–«оператора» похоже на тело подпрограммы–«выражения» с тем лишь исключением, что *имя–подпрограммы* не может играть роль переменной и появляется только в заголовке.

Чтобы использовать подпрограмму–«оператор» в программном модуле, применяют оператор *CALL* (вызвать), указывая в нем *имя–подпрограммы* и, если необходимо, список фактических параметров. Примеры:

```
CALL PAGE
CALL IMPCAR ('%', 72)
```

Действие оператора *CALL* эквивалентно выполнению тела соответствующей подпрограммы при условии подходящей замены формальных параметров фактическими.

IV.3.1.3. Замечания о подпрограммах ФОРТРАНа

а) **Формальные параметры** подпрограммы могут быть объявлены простыми переменными любого разрешенного ФОРТРАНОМ типа или массивами произвольного типа. В этом последнем случае можно – объявить одну или несколько границ как «переменные», например

```
SUBROUTINE MACHIN (TAB, M,N) REAL TAB (M,N)
...
END
```

Все непостоянные границы, как *M* и *N* в приведенном примере, должны быть в числе формальных параметров. Смысл таких параметров будет виден в IV.4.

б) **Результат** подпрограммы–«выражения» может иметь любой тип, но не может быть массивом.

в) Подпрограмма может содержать любое число операторов *RETURN*. Тем не менее весьма рекомендуется включать в подпрограмму только один такой оператор, размещая его в последней позиции (непосредственно перед *END*): гораздо проще, как мы это видели в гл. III**,** понять ход выполнения программы со схемой «1 вход – 1 выход». Из тех же соображений стараются избегать применения директивы *ENTRY*. Не упоминавшаяся до сих пор, эта директива позволяет задавать различные «точки входа» в подпрограмму. Она всегда успешно заменяется (когда она не служит ширмой для фокусов, которые заставляют содрогаться всякого порядочного программиста) структурой типа **выбрать ... иначе** ..., например в ФОРТРАНе – индексированным ветвлением, расположенным в начале подпрограммы.

IV.3.2. Подпрограммы в АЛГОЛе W

АЛГОЛ W содержит особенно элегантный механизм обработки подпрограмм, называемых здесь **«процедурами».**

Процедура определяется **объявлением процедуры,** которая синтаксически играет ту же роль, что и обычные объявления (переменных, массивов), и появляется, следовательно, в начале программы или, вообще говоря, в начале блока, после *BEGIN,* но перед первым оператором (III.5.1). Объявленное таким образом имя процедуры подчиняется обычным правилам структуры блоков и, значит, может быть идентификатором другого объекта (процедуры, переменной, массива) во внутреннем или непересекающемся блоке. Кроме того, как мы увидим, тело процедуры может содержать блок, следовательно, в частности, объявления процедур, которые локальны в этом блоке.

Подпрограмма–«оператор» имеет заголовок объявления в виде

PROCEDURE имя–процедуры (объявление форм–параметра–1, ...,
объявление форм–параметра–n);

или

PROCEDURE имя–процедуры;

Тело такой процедуры есть просто произвольный оператор АЛГОЛа W (составной или обычный), который может работать с объявленными формальными параметрами.

Объявления форм–параметров позволяют уточнить имена и типы параметров, а также «способ передачи», который может быть *VALUE* (аргумент), *RESULT* (результат), *VALUE RESULT* (модифицируемый параметр) или может отсутствовать. Точный смысл указателей способа передачи будет дан в следующем разделе; просим читателя временно не обращать на них внимание в производимых здесь примерах.

Вот объявление процедуры–«оператора» АЛГОЛа W:

```
АЛГОЛ W

PROCEDURE IMPRIMECARAC (STRING (1) VALUE CARAC;
                        INTEGER VALUE POSITION);
COMMENT:    НАПЕЧАТАТЬ СТРОКУ ИЗ 120 ЛИТЕР, СОДЕРЖАЩУЮ
            ЛИТЕРУ CARAC В СТОЛБЦЕ С НОМЕРОМ "POSITION" И
            ПРОБЕЛЫ В ДРУГИХ ПОЗИЦИЯХ. ЕСЛИ СПЕЦИФИКАЦИЯ
            НЕВЕРНА, ПЕЧАТАЮТСЯ ЗВЕЗДОЧКИ ВО ВСЕЙ СТРОКЕ;
            BEGIN
                STRING (120) LIGNE;
                IF (POSITION) >= 1) AND (POSITION <= 120)
                THEN
                        BEGIN
                        LIGNE := " "; COMMENT:
                        120 ПОЗИЦИЙ
                        ЗАПОЛНЕНЫ ПРОБЕЛАМИ;
                        LIGNE (POSITION – I|1) := CARAC
                        END
                ELSE FOR I := 0 UNTIL 119 DO
                        LIGNE (I|1); = '*';
                WRITE (LIGNE)
            END IMPRIMCARAC
```

Замечание: Благодаря соглашению в АЛГОЛе, по которому всякий идентификатор, следующий за *END* в той же строке, воспринимается как комментарий, можно улучшить читаемость программы, повторив ее имя после конечного *END* здесь *END IMPRIMCARAC*.

Подпрограмма–«выражение» имеет заголовок вида

тип имя–процедуры (объявление форм–параметра–1, ...,
объявление форм–параметра–n);

где *тип* – это тип значения, выдаваемого процедурой.

Тело такой процедуры – это произвольное выражение АЛГОЛа W. В АЛГОЛе W, как это уже отмечалось, понятие выражения было расширено так, чтобы можно было обозначать значения, которые являются результатом сложного вычисления: выражения–блоки, условные выражения, выражения *CASE*.

Таблица на следующей странице резюмирует эти расширения и соответствия «оператор–выражение» в АЛГОЛе W. Здесь $i_1,...,i_n$ означают любые операторы; $d_1,..., d_m$

– объявления; $e_1,\ldots, e_m$ – выражения; *лог* – логическое выражение; *цел* – целое выражение.

Вот два примера процедур–«выражений»:

АЛГОЛ W

```
INTEGER PROCEDURE MINIMUM (INTEGER VALUE X, Y)
        COMMENT: НАИМЕНЬШЕЕ ИЗ ДВУХ ЗНАЧЕНИЙ;
        IF X <= Y THEN X ELSE Y
```

АЛГОЛ W

```
LONG REAL PROCEDURE RACINE (LONG REAL VALUE X, EPS);
            COMMENT: RACINE–КОРЕНЬ;
            COMMENT: ВЫЧИСЛЕНИЕ КВАДРАТНОГО КОР–
            НЯ ИЗ X С ТОЧНОСТЬЮ EPS ПОСЛЕДОВАТЕЛЬ–
            НЫМИ ПРИБЛИЖЕНИЯМИ В СЛУЧАЕ НЕВЕР–
            НЫХ АРГУМЕНТОВ ВЫДАЕТСЯ X;
            IF (X < = 0) THEN X
            ELSE
                    BEGIN
                    LONG REAL A,B;
                    A : = 0;
                    B: = IF X <= I THEN 1 ELSE X;
                    WHILE B – A > 2*EPS DO
                        BEGIN
                        COMMENT: ИНВАРИАНТ ЦИКЛА:
                        A < = КВАДРАТНЫЙ КОРЕНЬ ИЗ X = B;
                        LONG REAL MILIEU;
                        MILIEU : = (A + B)/2
                        IF MILIEU ** 2 < X THEN
                                    A : = MILIEU
                        ELSE B : = MILIEU
                        END;
                    COMMENT: НИЖЕ–ВЫДАЧА ЗНАЧЕНИЯ;
                    (A + B)/2
                    END RACINE
```

(Цикл *WHILE* заканчивается только, если *EPS* превосходит некоторое положительное значение, зависящее от машины; ср. II.1.1.5 и упражнение II.2.)

Чтобы вызвать подпрограмму в АЛГОЛе W, пишут ее имя, за которым, если необходимо, следует заключенный в скобки список фактических параметров, имеющих типы, совместимые с типами соответствующих формальных параметров. Таким образом, получается выражение, если подпрограмма имеет тип «выражение», и оператор – в противном случае.

Примеры: *MINIMUM(X,3*Y + 2) IMPR1MCARAC("Z", 41)*

Вызов процедуры может включаться в зависимости от структуры блока либо в блок, где эта процедура объявлена, либо во внутренний блок. Это предполагает, что подпрограмма образует нечто целое, анализируемое транслятором как единый модуль. Тем не менее в АЛГОЛе W можно транслировать отдельно программы и процедуры; мы не будем здесь уточнять подробности правил, позволяющих включать в программный модуль обращения к процедурам, объявленным в другом, отдельно транслируемом модуле.

СООТВЕТСТВИЕ ОПЕРАТОРЫ-ВЫРАЖЕНИЯ В АЛГОЛе W

УПРАВЛЯЮЩАЯ СТРУКТУРА		СОСТАВНОЙ ОПЕРАТОР		СОСТАВНОЕ ВЫРАЖЕНИЕ	
		Синтаксис	Действие	Синтаксис	Значение
ЦЕПОЧКА		(оператор-)блок *BEGIN* $d_1; d_2; \ldots; d_m;$ $i_1; i_2; \ldots; i_n;$ *END*	Выполнение i_1, затем $i_2, \ldots$, затем i_n; возможно с помощью объектов, введенных объявлениями $d_1, \ldots, d_m$.	(выражение-) блок *BEGIN* $d_1; d_2; \ldots, d_m;$ $i_1; \ldots, i_n;$ e_1 *END*	Значение e_1 после выполнения i_1, затем $i_2, \ldots$, затем i_n, возможно с помощью объектов, введенных объявлениями $d_1, \ldots, d_m$
ЦИКЛ	ЦИКЛ "ПОКА"	*WHILE лог DO* i_1	повторение i_1 пока *лог* истинно	(использовать, если необходимо, выражение-блок)	
	ЦИКЛ СО СЧЕТЧИКОМ	*FOR* $i := e_1$ *STEP* e_2 *UNTIL* e_3 *DO* i_1	повторение i_1 с параметром i, меняющимся от e_1 до e_3 с шагом e_2 (эти три выражения - целые)		
ВЕТВЛЕНИЕ	АЛЬТЕРНАТИВА (a)	*IF лог THEN* i_1 *ELSE* i_2	Выполнение i_1, если *лог* есть *истина* и i_2 в противном случае	*IF лог THEN* e_1 *ELSE* e_2	e_1, если *лог* есть *истина*, e_2 в противном случае
	АЛЬТЕРНАТИВА (b)	*IF лог THEN* i_1	Выполнение i_1, если *лог* есть *истина*		
	МНОГОЗНАЧНЫЙ ПЕРЕКЛЮЧАТЕЛЬ	*CASE цел OF* *BEGIN* $i_1;$ $i_2;$ ..... i_n *END*	для целого значения j выполнение i_j, если $1 \leqslant j \leqslant n$, в противном случае - ошибка	*CASE цел OF* (e_1, e_2, ... e_n)	для целого значения *цел* j значение e_j, если $1 \leqslant j \leqslant n$, в противном случае - ошибка

IV.3.3. Подпрограммы в ПЛ/1

Подпрограммы в ПЛ/1 объявляются в виде «блоков–процедур» с предшествующей *меткой,* которая служит именем процедуры:

```
имя–подпрограммы: PROCEDURE...;
                  оп–1;
                  оп–2;
                  ...
                  оп–n
                  END имя–подпрограммы;
```

Первая строка (здесь неполная) служит заголовком подпрограммы; можно вслед за конечным *END* указывать имя соответствующей подпрограммы (здесь *имя–подпрограммы),* предложение факультативное, но рекомендуемое.

Слово *PROCEDURE* в объявлении сопровождается списком параметров, если они есть; тип процедуры и типы параметров объявляются, если это необходимо, в теле процедуры:

ПЛ/1

```
IMPCAR: PROCEDURE(CARAC, N);
        DECLARE CARAC CHARACTER (1),
                N BINARY FIXED (15, 0);
        /*  НАПЕЧАТАТЬ СТРОКУ ИЗ 120 ЛИТЕР. СОДЕРЖАЩУЮ ЛИТЕРУ
            CARAC В ПОЗИЦИИ N И ПРОБЕЛЫ В ДРУГИХ ПОЗИЦИЯХ;
            ЕСЛИ N НЕПРАВИЛЬНОЕ, ТО НАПЕЧАТАТЬ СТРОКУ
            ЗВЕЗДОЧЕК */
        DECLARE LIGNE CHARACTER (120);
        IF N >= 1 AND N <= 120 THEN
            DO
            LIGNE =' '/* ДОПОЛНЯЕТСЯ 119 ПРОБЕЛАМИ */;
            SUBSTR (LIGNE, N, 1) = CARAC;
            END;
        ELSE DO 1 = 1 TO 120;
            SUBSTR (LIGNE, I, 1) = '*';
            END;
        PUT LIST (LIGNE);
        END IMPCAR;
```

Если подпрограмма имеет тип «выражение», то тип выдаваемого результата указывается в заголовке процедуры атрибутом *RETURNS* (тип); например,

MINIMUM : PROCEDURE (X, Y) RETURNS (BINARY FIXED (15, 0));

Такая подпрограмма должна кончаться выполнением оператора

RETURN (выражение)

где *выражение* – это выражение соответствующего типа, значение которого будет результатом подпрограммы.

ПЛ/1

```
MINIMUM: PROCEDURE (X,Y) RETURNS (BINARY FIXED);
        IF X < Y THEN RETURN(X);
        ELSE RETURN(Y);
        END MINIMUM;
```

Можно опускать предложение

RETURNS (mun)

в заголовке процедуры, если тип ее может быть определен по умолчанию (так, например, для процедуры *MINIMUM,* имя которой начинается с M, тип вырабатываемого ею значения неявно рассматривается как *REAL BINARY FIXED (15, 0),* что совпадает с *BINARY FIXED*). Само собой разумеется, что использовать эту возможность рекомендуется еще меньше, чем в ФОРТРАНе, где подпрограммы явно обозначаются как *SUBROUTINE* или *FUNCTION*.

Подпрограмма «оператор» или «выражение» может содержать, произвольные объявления и операторы, в частности блоки *BEGIN; ... ; END;* или блоки–процедуры *METKA: PROCEDURE ...; ...; END;*

Точно так же, как в АЛГОЛе W, можно конструировать процедуры, обладающие локальными процедурами, и части программ, откуда можно вызвать процедуру, определяются обычными правилами структуры блока; можно также использовать процедуру, транслировавшуюся отдельно (такие процедуры будут иметь атрибут *EXTERNAL*).

IV.4. Способы передачи информации между программными модулями

IV.4.1. Проблема

Проблема, к изучению которой мы сейчас приступаем, а именно проблема передачи информации от одной программы к другой, традиционно представляет собой одну из наименее известных программистам и одну из проблем, которые служат источником наибольшего количества недоразумений.

Определения языков программирования и, следовательно, учебники этих языков долгое время оставляли эти проблемы в тени, так что лучшим ответом на различные возникавшие вопросы было попробовать и посмотреть, что «получается» у машины. ФОРТРАН – типичный в этом отношении язык; создатели ПЛ/1 поставили задачу более четко, но выбранное решение далеко от желаемой простоты и ясности. Более удовлетворительный ответ был предложен АЛГОЛом 60 и улучшен АЛГОЛом W, но и в этом случае остается еще достаточно вопросов, связанных с физическим представлением на конкретных машинах. Такие языки, как ПАСКАЛЬ и АЛГОЛ 68, не предлагают ничего нового, кроме синтаксических упрощений. Действительно, только совсем недавние разработки начали давать по–настоящему удовлетворительные решения.

Проблема состоит в том, чтобы ключевые слова аргумент, результат и модифицируемый параметр, с помощью которых мы определяли различные типы параметров, действительно соответствовали физической реальности передачи параметров между программными модулями: объект, передаваемый как «аргумент», должен сохранять неизменным свое значение; «результат» должен быть эффективно вычислен и передан в вызывающую программу; «модифицируемый параметр» должен

быть изменен вызываемой программой так, как программа это предусматривает.

Речь идет о том, чтобы, уточнив один раз эти принципиальные определения, изучить, как они реализуются в обычных языках программирования. Действительно, многие из обсуждаемых вопросов вращаются вокруг проблемы:

Как воздействует на передаваемый подпрограмме фактический параметр возможная модификация значения соответствующего формального параметра в теле подпрограммы?

Ответы на этот вопрос станут яснее, если передаваемый вызываемой программе фактический параметр рассматривать как *механизм доступа* к некоторой информации, принадлежащей вызывающей программе. В простейшем случае этот механизм состоит в выдаче значения информации и ее адреса; он может быть также некоторым достаточно сложным алгоритмом, используемым при каждом доступе к передаваемой информации.

Пример

Пример прояснит проблему. Пусть надо определить наибольшее из абсолютных значений двух вещественных x и y. Решение записывается в виде

```
программа максабсзнач: ВЕЩ (аргументы a, b : ВЕЩ)
    {максимум абсолютных значений a и b}
    переменные x, y : ВЕЩ;
    x ← если a ≥ 0 то a иначе –a;
    y ← если b ≥ 0 то b иначе –b;
    максабсзнач ← если x > y то x иначе y
```

Теперь пусть по каким–либо соображениям желательно отказаться от объявления двух локальных переменных x и y; можно заметить, что результат подпрограммы не меняется, если переписать ее таким образом:

```
программа максабсзнач: ВЕЩ (аргументы a, b : ВЕЩ)
    {максимум абсолютных значений a и b}
    a ← если a ≥ 0 то a иначе –a;
    b ← если b ≥ 0 то b иначе –b;
    максабсзнач ← если a > b то a иначе b
```

Проблемы: совместимы ли присваивания формальным параметрам a и b с их объявлениями в качестве аргументов? Если да, что происходит, когда программа использует максабсзнач следующим образом (u, v и w – вещественные переменные):

```
w ← максабсзнач (u, v)
```

Как это влияет на u и v? Каков эффект действий

```
w ← максабсзнач (u + v, u – v)
```

или `w ← максабсзнач (u, –7.68)`?

Чтобы ответить на эти вопросы, важно уточнить понятия **аргумент**, **результат** и **модифицируемый параметр**.

Мы сделаем это, говоря о *чистом чтении, чистой записи* и *чтении и записи* соответственно. К этим способам передачи параметров следует добавить еще один способ обмена данными между программными модулями, который не включает явных обозначений параметров: *обобществление данных,* которое приводит к объявлению некоторых объектов доступными в нескольких программных модулях без явной передачи при каждом вызове (IV.5). Интуитивный смысл этих различных методов показан на Рис. IV.3.

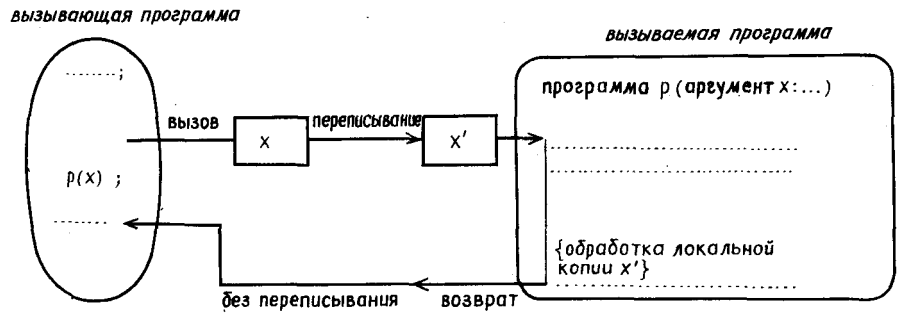


Рис. IV.3 Способы обмена информацией.

Все, что здесь обсуждается, можно непосредственно применять в случае, когда формальный параметр может обрабатываться в программе как переменная: фактический параметр должен быть, таким образом, переменной, элементом массива или «приравненным» объектом (например, формальным параметром типа «переменная» в вызывающей программе). Будем называть такой параметр *простым.* Отдельно будет рассмотрен случай массивов (IV.4.5), хотя единственное различие будет касаться физического представления; наконец, в IV.4.6 будет обсужден случай параметров, которые являются именами подпрограмм.

IV.4.2. Чистое чтение: передача значением

При **передаче «чистым чтением»** подпрограмма интересуется только значениями фактических параметров в момент вызова, но не может менять значения этих параметров в вызывающей программе.

Практически часто бывает полезным уметь обрабатывать формальные параметры как переменные в подпрограмме и, в частности, модифицировать их значения; все, что здесь требуется, чтобы эти модификации не влияли на фактические параметры в вызывающей программе. Решение, называемое *передачей значением,* состоит в «переписывании» начального значения фактических аргументов в локальную область подпрограммы, с которой будут связаны формальные аргументы. В этом случае все операции над этими формальными аргументами будут законны, поскольку они не влияют на фактические аргументы (Рис. IV.4).

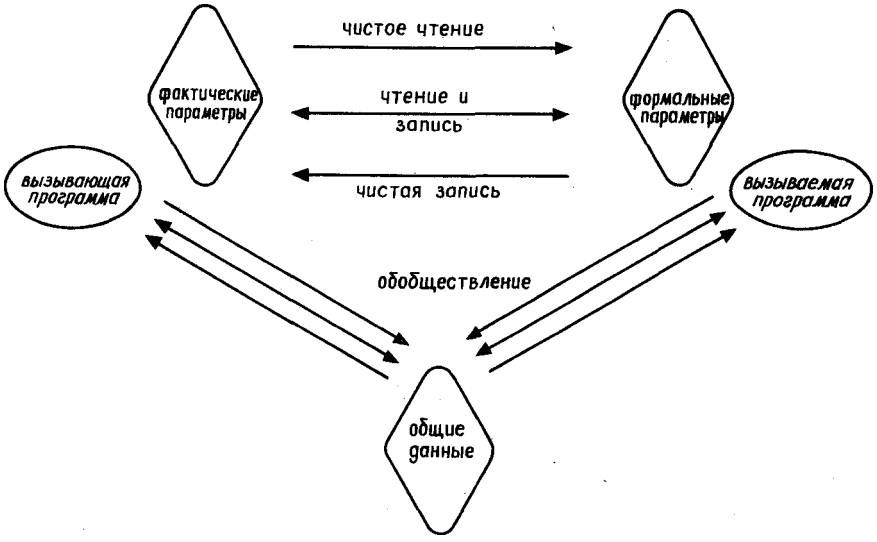


Рис. IV.4 Передача значением.

В передаче «значением» формальный аргумент обрабатывается как локальная переменная в теле подпрограммы, инициализируемая в начале каждого выполнения подпрограммы значением соответствующего фактического аргумента.

Далее мы будем предполагать, что всякий простой параметр, указанный в качестве аргумента, действительно передается значением. При таком способе передачи вторая версия подпрограммы максабсзнач теперь верна:

```
программа максабсзнач: ВЕЩ (аргументы a, b : ВЕЩ)
        {передача значением}
        если a < 0 то a ← –a;
        если b < 0 то b ← –b;
        максабсзнач ← если a ≥ b то a иначе b
```

Точно так же правомерны вызовы:

максабсзнач (u, v)
максабсзнач (u + v, u – v)
максабсзнач (u, –7.65)

В самом деле, подпрограмма оперирует только с локальными величинами; фактические параметры при этом не затрагиваются.

Из наших трех языков только АЛГОЛ W позволяет указать, что формальный параметр должен быть передан значением: достаточно использовать указатель способа передачи *VALUE*. Например,

АЛГОЛ W

```
LONG REAL PROCEDURE MAXVALABS (LONG REAL VALUE A,B);
    COMMENT:    СРЕДСТВО (НЕСКОЛЬКО СТРАННОЕ) ВЫ–
                ЧИСЛИТЬ НАИБОЛЬШЕЕ ИЗ ДВУХ АБСО–
                ЛЮТНЫХ ЗНАЧЕНИЙ A И B;
    BEGIN
    IF A < 0 THEN A := –A;
    IF B < 0 THEN B := –B;
    IF A >= B THEN A ELSE B
    END MAXVALABS
```

В ПЛ/1 способ передачи значением может быть указан в момент вызова, т.е. на уровне фактического параметра. Действительно, только константы и выражения (в противоположность переменным и элементам массивов) передаются значениями. Так, для программы

ПЛ/1

```
MAXVALABS: PROCED U RE (A,B) RETURNS BINAR Y FLOAT (53);
           DECLARE (A,B) BINARY FLOAT (53);
           IF A < 0 THEN A = –A;
           IF B < 0 THEN B = –B;
           IF A > B THEN RETURN A; ELSE RETURN B;
           END MAXVALABS;
```

следующие вызовы будут соответствовать передаче значением

MAXVALABS (46.E0, –52.E1) (результат *52.E1*)

MAXVALABS (U + V, U – V)

(*U* и *V* – это переменные *BINARY FLOAT(53)*). Напротив, если написано

W = MAXVALABS (U, 7.65E0)

U будет, возможно, преобразовано в *– U* после этого вызова. Заметим, однако, что можно реализовать эффект передачи значением для переменной или элемента массива: достаточно превратить их в выражения, заключив в скобки

```
W = MAXVALABS((U), 7.65E0)
```

До сих пор с условностями ПЛ/1 можно согласиться. К сожалению, они сопровождаются следующей оговоркой: если атрибуты формального и фактического параметров не идентичны, т.е. должно иметь место *преобразование типов,* выполняется *ередача значением.* Это абсурдно! Из–за богатства атрибутов ПЛ/1, в частности атрибутов числовых, два объекта зачастую отличаются второстепенными атрибутами. Так, в

```
DECLARE U BINARY FLOAT (53),
        V BINARY FLOAT (52);
U, V = –7.65E0;
PUT LIST (MAXVALABS (U, V));
```

теоретически будет иметь место преобразование для *V,* но не для *U.* Тогда *U* будет равно –7.65 после вызова *MAXVALABS,* а *V* будет иметь значение +7.65.

Программист, создающий программу широкого применения, не имеет в ПЛ/1 никакой гарантии, что тип фактического параметра может быть сохранен при передаче. Поэтому в целях обеспечения эффекта передачи значением и «защиты данных» программист должен сам запрограммировать локальное копирование – как в ФОРТРАНе. В ФОРТРАНе можно добиться эффекта передачи значением, только используя явно инициализируемую локальную переменную:

ФОРТРАН

```
      DOUBLE PRECISION FUNCTION MAXABS (A,B)
      DOUBLE PRECISION A,B
C     ВЫЧИСЛЕНИЕ МАКСИМУМА АБСОЛЮТНЫХ ЗНАЧЕ–
C     НИЙ А И В С ПОМОЩЬЮ ПЕРЕДАЧИ ЗНАЧЕНИЕМ
      DOUBLE PRECISION X,Y
      X = A
      Y = B
      IF (X.LT.0) X = –X
      IF (Y.LT.0) Y = –Y
      MAXABS = X
      IF (Y.GT. MAXABS) MAXABS = Y
      RETURN
      END
```

Забывая переписать параметр аргумент, программист совершает типичную и серьезную ошибку в ФОРТРАНе; вопрос «нужна ли передача значением?», как хороший рефлекс, должен ставиться систематически для каждого аргумента подпрограммы.

Во всех типах передачи параметров подпрограмма способна воздействовать на фактические параметры. Передача значением соответствует простому и надежному понятию «безвозвратно» передаваемого параметра. Существенная часть ошибок, встречающихся в многомодульных программах, связана с недопустимыми модификациями параметров, передаваемых подпрограммами. Таким образом, передача значением представляет собой важное средство повышения надежности программ, применяемое всегда, когда это возможно.

IV.4.3. Чистая запись, передача результата

Ситуацией, симметричной по отношению к передаче чистым чтением, является передача **«чистой записью»:** это случай, когда фактический параметр должен получить значение вследствие вызова подпрограммы; его начальное значение безразлично. Эта ситуация имеет место для результата подпрограммы–«выражения».

Практически надо будет трактовать соответствующий объект как переменную в подпрограмме перед тем, как выдать конечное значение этой переменной фактическому параметру, приняв соглашение, что ее начальное значение не определено при каждом вызове. Решение, называемое *передачей результата,* состоит в использовании для формального параметра некоторой локальной, не инициализируемой при вызове области в подпрограмме и «переписывании» из нее конечного значения в фактический параметр в момент возврата (Рис. IV.5).

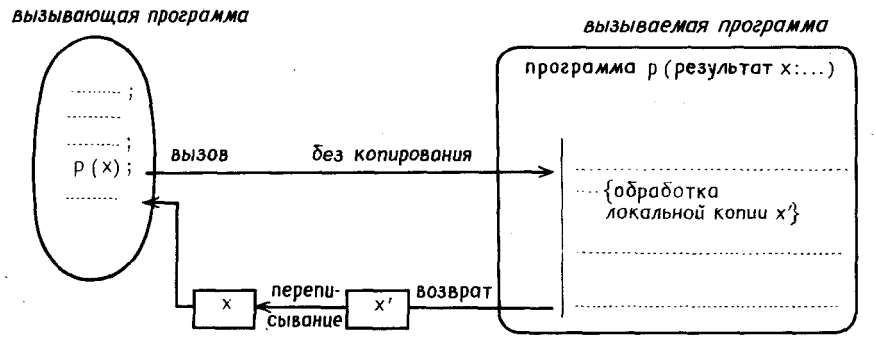


Рис. IV.5 Передача результата.

При передаче результата формальный параметр трактуется в подпрограмме как локальная переменная, начальное значение которой не определено, а конечное значение присваивается соответствующему фактическому параметру.

Важно отметить, что фактический параметр, соответствующий формальному параметру–«результату», является обязательным объектом вызывающей программы, которому можно присвоить значение: переменная, элемент массива или приравненный к ним объект. Использование константы или выражения в качестве фактического параметра – это ошибка.

В дальнейшем будем предполагать, что этот способ передачи действительно употребляется для всякого простого параметра, специфицированного как результат, и для результата подпрограммы–«выражения».

Заметим, что иногда предпочтительно употреблять подпрограмму–«выражение», выдающую единственный результат «составного» типа (гл. V), чем подпрограмму с несколькими параметрами–**результатами**.

Из наших трех языков **только АЛГОЛ W** предлагает передачу результата; соответствующий указатель имеет обозначение *RESULT*. Например,

АЛГОЛ W

```
INTEGER PROCEDURE EQUA2
        (LONG REAL VALUE A, B, C;
        LONG REAL RESULT XI, X2);
        COMMENT: РЕШЕНИЕ УРАВНЕНИЯ ВТОРОЙ СТЕПЕ–
        НИ: AX2 + BX + C = 0.
        РЕЗУЛЬТАТ ПРОЦЕДУРЫ–ЭТО ЦЕЛЫЙ КОД:
              2    ЕСЛИ ОБА КОРНЯ ВЕЩЕСТВЕННЫЕ,
              1    ЕСЛИ КОРЕНЬ ЕДИНСТВЕННЫЙ (ИЛИ
                   ДВОЙНОЙ),
              0    ЕСЛИ НЕТ ВЕЩЕСТВЕННЫХ КОРНЕЙ,
              –1   ЕСЛИ ВСЯКОЕ ВЕЩЕСТВЕННОЕ ЕСТЬ
                   КОРЕНЬ.
        ЕСЛИ КОРНИ ЕСТЬ, ТО X1 – ПЕРВЫЙ КОРЕНЬ,
        X2 – ВТОРОЙ;
        IF A ¬=0 THEN
             BEGIN LONG REAL DELTA;
             DELTA := B**2 – 4.*A*C;
             IF DELTA < 0 THEN 0
             ELSE IF DELTA = 0 THEN
                  BEGIN
                  X1 := –B/(2.*A);
                  1
                  END
              ELSE
                   BEGIN
                   COMMENT: LONGSQRT(X) – ЭТО КО–
                   РЕНЬ ИЗ X;
                   X1 := (–B–L0NGSQRT(DELTA))/(2.*A);
                   X2 := (–B + LONGSQRT(DELTA))/(2.*A);
                   2
                   END
        END
        ELSE COMMENT : ВЫРОЖДЕННЫЙ СЛУЧАЙ;
             IF B ¬= 0 THEN
                  BEGIN
                  X1 := –C/B
                  1
                  END
             ELSE IF C = 0 THEN –1 ELSE 0
```

Мы широко использовали в этой процедуре смешанные выражения, сочетающие целые и *LONG REAL,* когда нет возможных двусмысленностей.

Заметим, что представленный алгоритм переписанной из математического анализа задачи, которая известна со школьной скамьи, очень важен с точки зрения машинной реализации. В частности, сравнения с точным нулем имеют не очень много смысла; кроме того, формула $(-b \pm \sqrt{b^2 - 4ac})/2a$ может вызвать неприятности в системе, связанной с машинным представлением вещественных чисел (II.1.1.5). Построение удовлетворительного алгоритма для квадратного уравнения – не тривиальная задача: мы касаемся здесь только проблемы передачи параметров.

Процедура *EQUA2* – типичная подпрограмма–«выражение» в АЛГОЛе W: она принципиально использует выражения–блоки и условные выражения. Тело процедуры является условным выражением

IF c THEN e_1 ELSE e_2

где e_1 и e_2— выражения–блоки, составленные в свою очередь из условных выражений, использующих выражения–блоки.

В ФОРТРАНе для изображения передачи результата нет необходимости использовать локальную переменную, как в случае передачи значением: достаточно непосредственно обращаться с формальным параметром как с локальной переменной. Это делается так же, как в ПЛ/1. Использование локальных переменных может, однако, быть полезным из соображений эффективности (защита «локальности обращений» в виртуальной памяти).

IV.4.4. Чтение и запись: значение–результат, адрес, имя

Передача модифицируемыми параметрами, или «чтение и запись», позволяет подпрограмме использовать и менять при желании значения фактических параметров. Такая ситуация дает больше гибкости, но она же и более опасна: доказано, действительно, что тем легче обеспечить надежность программы, чем больше ограничены обмены информацией между программными модулями.

Этот способ передачи может осуществляться разными путями, которые дают примерно одинаковый результат (но только примерно): *передача по адресу, передача значения–результата, передача по имени.* Во всех случаях ясно, что фактический параметр может быть, как в случае передачи результата, только переменной или приравненным объектом.

Когда мы будем специфицировать модифицируемый параметр, будем считать, что имеет место один из этих трех способов: в ситуациях, встречающихся в дальнейшем, все они дают один и тот же результат. Это, однако, не всегда верно, как свидетельствует пример из упражнения IV.1.

а) *Передача значения–результата*

Передача «значения–результата» есть комбинация передачи значением и передачи результата: подпрограмма обрабатывает формальный параметр как локальную переменную; чтобы обеспечить эффект «чтения и записи», начальное значение этой локальной переменной становится начальным значением фактического параметра, а ее конечное значение присваивается фактическому параметру в момент возврата.

При **передаче «значения–результата»** формальный параметр рассматривается как локальная переменная в теле подпрограммы, принимающая в момент каждого вызова значение соответствующего фактического параметра; конечное значение этой переменной присваивается фактическому параметру после каждого выполнения подпрограммы.

Таким образом, в этом случае имеют место и копирование начального значения, и обратное переписывание конечного значения (Рис. IV.6).

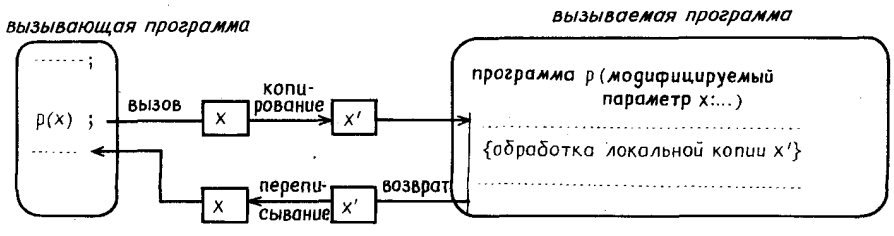


Рис. IV.6 Передача значения–результата.

Чтобы использовать этот способ передачи в ПЛ/1, программист должен сам объявить локальную переменную, соответствующую параметру, и включить в начало и

конец подпрограммы копирование и обратное переписывание параметра. Так же обстоит дело и в ФОРТРАНе, за исключением некоторых систем, предпочитающих этот способ передаче по адресу (ср. ниже).

В АЛГОЛе W передача «значение–результат» специфицируется указателем способа передачи

VALUE RESULT

Пример:

```
АЛГОЛ W

PROCEDURE ZAMENA (STRING (80) VALUE RESULT TEXTE,
        STRING (1) VALUE CARAC_1, CARAC_2);
COMMENT:   ЭТА ПРОЦЕДУРА ЗАМЕНЯЕТ В ТЕКСТЕ
           ВСЕ ВХОЖДЕНИЯ ЛИТЕРЫ CARAC–1 НА
           CARAC–2;
                FOR I := 0 UNTIL 19 DO
                     IF TEXTE (I|1) = CARAC_1 THEN
TEXTE (I|1) := CARAC_2
```

Заметьте, что

```
STRING (80) T;
ZAMENA (T, "A", "B")
```

эквивалентно

```
STRING (80) T;
T := ZAM (T, "A", "B")
```

где *ZAM* – процедура–«выражение»:

```
АЛГОЛ W

STRING (80) PROCEDURE ZAM
           (STRING (80) VALUE TEXTE;
           STRING (I) VALUE CARAC_1, CARAC_2);
      BEGIN
      FOR I: = 0 UNTIL 79 DO
            IF TEXTE (I|1) = CARAC_1 THEN
                  TEXTE (I|1) := CARAC–2;
      TEXTE
END ZAM
```

б) *Передача по адресу (или по «ссылке»)*

В отличие от других способов передачи, определяемых внешним способом относительно воздействия на параметры, концепция *передачи по адресу* использует понятие физического расположения программы.

В этом способе при каждом вызове передается не значение, а адрес фактического параметра. Таким образом, подпрограмма получает доступ к самому фактическому параметру, а не к некоторой локальной копии.

При **передаче по «адресу»** формальный параметр обрабатывается как переменная, адрес которой, передаваемый в момент каждого вызова, есть адрес соответствующего фактического параметра.

Этот способ передачи обычен для ФОРТРАНа и ПЛ/1; в АЛГОЛе W его нельзя использовать для простых параметров. С точки зрения воздействия на фактические параметры он эквивалентен, вообще говоря, передаче значения–результата; исключения составляют ситуации, когда для одной и той же переменной имеют место одновременно и передача по адресу, и обобществление (см. упражнение IV.1).

Когда задана передача параметра по адресу, подпрограмма имеет прямой доступ к параметру благодаря адресу, переданному вызывающей программой. Таким образом, речь идет о способе, довольно опасном в том смысле, что трудно обеспечить «защиту» переменных программного модуля от возможных искажений в другом модуле.

Когда фактический параметр является константой или сложным выражением, передаваемый адрес – это адрес области памяти, содержащей значение константы или выражения. Это может приводить к курьезным явлениям, когда подпрограмма пытается изменить значение формального параметра. Пусть есть подпрограмма

ФОРТРАН

```
      SUBROUTINE PLUS (I)
C     УВЕЛИЧЕНИЕ ЗНАЧЕНИЯ АРГУМЕНТА НА 1
      I = I + 1
      RETURN
      END
```

Пусть A, B, C – целые переменные. Вызов

CALL PLUS(A)

увеличит значение *A* на 1. Вызов

*CALL PLUS (2*A + 3*B + C)*

увеличит на 1 содержимое области памяти, выделенной для целого значения и временно используемой для размещения значения выражения. Действие этого вызова в вызывающей программе имеет обычно нулевой эффект. Что касается вызова

CALL PLUS (0)

его действие состоит в размещении единицы в области, содержащей предварительно значение 0. Итак, если такой метод используется транслятором, вполне может случиться, что эта область в равной мере используется всеми операторами программы, работающими с константой 0, например

IF (J.EQ.0) ...

Во всех последующих ссылках тогда будет использоваться 1 вместо 0, что может привести к неприятностям. Большинство программистов на ФОРТРАНе имели случай сами испытать последствия этого оригинального метода «вариации констант» (см. упражнение III.3, параметр *LA*).

Вызовы *PLUS*, подобные двум последним, очевидным образом являются ошибочными, но структура ФОРТРАНа, задуманного с целью систематически разрешать раздельную трансляцию подпрограмм, запрещает транслятору выявлять такие ошибки. Напротив, в языках блочной структуры, таких, как АЛГОЛ или ПЛ/1, когда подпрограмма объявляется «внутри» другого программного модуля, транслятор располагает необходимыми атрибутами для выполнения некоторого числа проверок.

Заметим, наконец, что применение передачи по адресу позволяет объяснить, почему в ПЛ/1 достаточно заключить переменную в скобки, чтобы реализовать эффект передачи по адресу: если *P* –подпрограмма (процедура), а *X* – переменная, то вызов

CALL P(X)

передает подпрограмме *P* адрес *X*, что позволяет работать непосредственно с переменной; напротив, при вызове

CALL P((X))

подпрограмме *P* передается адрес «выражения», инициализирующего значение *X* и размещенного отдельно в памяти; таким образом, всякое присваивание формальному параметру не воздействует на *X*.

в) *Передача именем*

Передача параметра *именем* (понятие, введенное АЛГОЛом 60) соответствует ситуации, в которой необходимо, чтобы замена формального параметра фактическим, имела такой же эффект, как настоящая *текстуальная подстановка* одного вместо другого в тексте подпрограммы.

Чтобы лучше разобраться в тонкостях этого понятия и его оригинальности по отношению к описанным ранее способам, рассмотрим подпрограмму p с двумя формальными параметрами i и j, каждый из которых участвует в присваиваниях:

```
программа p : ЦЕЛ (модифицируемые параметры i, j: ЦЕЛ)
    ...
    j ← 3; ...; i ← 4;
    ...
```

Рассмотрим теперь следующую программу, которая вызывает подпрограмму p:

```
переменные n : ЦЕЛ, m : ЦЕЛ;
массив a[1 : 100] : ЦЕЛ;
    ...
    n ← 5;
    m ← p(a[n + 2], n)
    ...
```

В момент вызова n + 2 равно 7. Во всех рассмотренных выше формах передачи параметров, если вызов правомерен, то присваивание

i ← 4

есть присваивание элементу a[7] или локальной копии этого элемента. Идея передачи именем состоит в том, что на протяжении всей подпрограммы первый параметр i должен «представлять» a[n + 2], а второй j «представляет» n. Последовательность операторов

j ← 3; i ← 4

должна поэтому иметь тот же эффект в нашем примере вызова, что и

n ← 3; a[n + 2] ← 4

В случае передачи по адресу или передачи значения–результата модифицировался бы элемент a[5], а не a[7].

Говорят, что параметр подпрограммы **передается именем**, если выполнение каждого оператора подпрограммы с участием формального параметра эквивалентно выполнению того же оператора при условии замены формального параметра на имя фактического параметра.

Если язык обладает блочной структурой в смысле АЛГОЛа (с локальной областью действия идентификаторов), применение предыдущего правила предполагает, что предварительно происходит изменение имен локальных переменных в подпрограмме, если они идентичны именам некоторых фактических параметров.

Последнее предложение означает, что если подпрограмма p из нашего примера обладает локальной переменной с идентификатором n, то n заменятся другим именем, например n', во всей подпрограмме до применения правила подстановки, чтобы исключить неоднозначность.

Конечно же, нет речи о том, чтобы требовать от транслятора действительной подстановки имен; требуется просто, чтобы он генерировал эквивалентный код. Решение (схематически) состоит в том, чтобы убедиться, что формальный параметр, передаваемый именем, действительно соответствует *подпрограмме доступа к фактическому параметру:* в нашем примере первый параметр – это подпрограмма, вычисляющая a[n + 2] в зависимости от n (и адреса a). Любая ссылка на j в теле подпрограммы p порождает новое вычисление a[n + 2] с помощью соответствующей «подпрограммы». Такой механизм позволяет «моделировать» текстуальную подстановку при выполнении без фактического ее осуществления при компиляции.

В наших трех языках только АЛГОЛ W предлагает передачу именем в форме АЛГОЛа 60. Это тот же «вариант по умолчанию», потому что параметр без любого из атрибутов *VALUE, RESULT* или *VALUE RESULT,* как *A* в

INTEGER PROCEDURE P (LOGICAL A)

передается именем.

Передача именем позволяет реализовать очень кратким (и иногда туманным) описанием относительно сложные функции. Так, обсуждаемая ниже подпрограмма *MAXMAT,* текст которой сам по себе говорит немного, позволяет после трех последовательных вызовов найти максимум одномерной матрицы, затем другой, двумерной матрицы, потом третьей матрицы пяти измерений.

Подпрограмма *MAXMAT* достаточно типична среди подпрограмм, использующих передаваемые именем параметры. Рассмотренная отдельно, она не имеет никакого смысла: если *ELEM* и *MAX* обрабатывались бы как переменные, то цикл *WHILE* повторял бы *SUP–INF+1* раз один и тот же условный оператор. Смысл становится очевидным, только когда известны фактические параметры: при первом вызове, например, *INDICE* «представляет» *I, ELEM* «представляет» *A(I), INF* и *SUP* – это 0 и 27, и цикл тогда представляет собой

```
WHILE I <= 27 DO
    BEGIN
    IF A(I) > MAX THEN MAX : = A(I);
    I := I + 1 END
```

(*I* инициализировано значением *INF,* например нулевым).

Этот вид обработки пользовался некоторой популярностью в момент появления АЛГОЛа 60. Однако передача именем достаточно скоро вышла из моды; программа типа той, что написана на следующей странице, в действительности очень неэффективна: каждое обращение к *ELEM* порождает здесь, например, в случае самого внутреннего вызова, включенного в вычисление *MAXC,* определение *I, J, K, L, M* (дважды), границ и адреса *C(I, J, K, L, M)*. И эта неэффективность не оправдывается даже ценой хорошей читаемости программы, скорее наоборот.

Почти все передачи именем могут быть сведены либо к случаям, когда передача значения–результата или передача по адресу были бы достаточны, либо к ситуациям, когда действительно необходима передача подпрограммы в качестве параметра (IV.4.6).

АЛГОЛ W

```
BEGIN
COMMENT: "ХИТРОУМНЫЙ" (?) СПОСОБ ВЫЧИСЛИТЬ МАК–
СИМУМ ЛЮБОГО ЦЕЛОГО МАССИВА;
INTEGER ARRAY A(0 :: 27);
INTEGER ARRAY B(–2::5, 1 :: 10);
INTEGER ARRAY C(0 :: 4, 0 :: 4, 0 :: 4, 0 :: 4, 0 :: 4);
INTEGER MAXA, MAXB, MAXC;
INTEGER I, J, K, L, M;
REAL PROCEDURE MAXMAT (INTEGER ELEM, INF, SUPJNDICE);
        BEGIN INTEGER MAX;
        COMMENT: MAXINTEGER–ЭТО ВСТРОЕННАЯ ПЕРЕМЕННАЯ В
                 АЛГОЛЕ W, ИМЕЮЩАЯ СВОИМ ЗНАЧЕНИЕМ
                 НАИБОЛЬШЕЕ ЦЕЛОЕ, ПРЕДСТАВИМОЕ В МАШИНЕ;
        MAX := –MAXINTEGER; INDICE : = INF;
        WHILE INDICE <= SUP DO
                BEGIN
                IF ELEM > MAX THEN MAX : = ELEM;
                INDICE := INDICE + 1
                END;
        MAX
        END MAXMAT;
...
... инициализация матриц A, B, C ...
...
MAXA := MAXMAT (A(1), 0, 27, I);
MAXB := MAXMAT (MAXMAT (B(I, J), 1, 10, J) –2, 5, I);
MAXC := MAXMAT (
          MAXMAT (
               MAXMAT(
                    MAXMAT (
                         MAXMAT (C(I, J, K, L, M), 0, 4, M),
                                 0, 4, L),
                         0, 4, K),
                    0, 4, J),
               0, 4, I);
WRITE ("МАКСИМУМЫ ТРЕХ МАТРИЦ :", MAXA,MAXB,MAXC)
END.
```

Например, нужно написать программу сортировки сравнениями (VII.3), не уточняя критерий сравнения, а полагая его формальным параметром типа ЛОГ, передаваемым именем. При вызове соответствующий фактический параметр мог бы быть

a[i] > a[j], где a – массив

или f(j + j) < f(i – j), где f – подпрограмма.

В этом случае предпочтительно сделать критерием параметр типа подпрограммы (ЦЕЛ × ЦЕЛ → ЛОГ).

Как бы то ни было, передача именем, редко необходимая практически, является концептуально важным средством – не потому ли, что оно является строгим определением способа передачи, которое начинающим часто кажется естественным: для них, действительно, если фактический параметр есть a[i, j], а формальный параметр m, всякое использование m в подпрограмме «представляет» a[i, j] с текущими

значениями i и j, даже если они изменились после момента вызова.

Макросы

По поводу передачи именем полезно отметить, что *текстуальная подстановка* – это механизм, реализующий макрооператоры (или макросы), имеющиеся в некоторых языках. Макрооператор – это приказ, адресуемый программе преобразования текста, действие которой логически воспроизводит действие транслятора (даже если физически эта программа объединена с транслятором) и осуществляет замену некоторых элементов текста программы («имена макросов») на тексты, заданные программистом («тела макросов»). Как подпрограмма макрооператор может иметь «формальные параметры», которые заменяются «фактическими параметрами» в момент включения в текст программы.

Путем некоторых модификаций мы могли бы заменить наш *MAXMAT,* например, макрооператором для программы обработки текста, которой подвергалась бы программа до того, как транслятор АЛГОЛа W приступает к работе.

Подчеркнем тот факт, что замена макрооператора (говорят также – *макрорасширения*) – это в полном смысле обработка текста, предшествующая трансляции, и что любое обращение к имени макроса порождает включение в программу тела соответствующего макроса. Макрооператоры, таким образом, приводят программиста к потере одного из преимуществ подпрограмм – осознания того, что текст подпрограммы транслируется только один раз, каково бы ни было число вызовов этой подпрограммы.

> Однако в случае подпрограмм, вызываемых из небольшого числа различных мест, макрооператоры дают интересное решение: они позволяют избежать потерь времени, порождаемых на многих машинах слишком частыми вызовами подпрограмм с сохранением всех преимуществ ясности и читаемости, вытекающих из хорошо организованного разделения программ на логические модули, в частности, в концепции «нисходящего» программирования (ср. гл.VIII, и в частности разд. VIII.3.3, где исследуются эти стратегические проблемы декомпозиции задач). Некоторые трансляторы, как, например, транслятор с языка ЛИС [Ишбия 75], разумно обрабатывают подпрограмму, вызываемую из единственного места, как макрооператор с переписыванием текста в место вызова.

Применение макроса позволит несколько обобщить возможности *MAXMAT*. В том виде, в котором подпрограмма записана выше, она позволяет вычислить максимум в абсолютно произвольном массиве, но не в чем–либо другом. Программа обработки макроса, входной язык которой не подчиняется строгим синтаксическим ограничениям обычных языков программирования, могла бы снять этот вид ограничений. Можно, таким образом, представить себе систему, позволяющую использовать макрос *MAXMAT,* один из формальных параметров которого соответствовал бы *способу перечисления* элементов множества (соответствующий фактический параметр мог бы быть, например, *FOR I := INF STEP 1 UNTIL SUP DO* или *FOR I := X, Y, Z, T, U DO*). Развитие необходимого формализма в использовании такой системы выходит за рамки этой книги; читателю, заинтересовавшемуся макросами, будет полезно разыскать книгу П. Д. Брауна, указанную в библиографии.

Системы обработки программ особенно употребительны на практике при программировании на ассемблерах; для многих языков этого типа, синтаксис которых очень строг, были созданы «макроассемблеры», делающие немного менее скучной работу программиста.

ФОРТРАН предлагает программисту возможность пользоваться конкретными функциями, соответствующими, вообще говоря, примитивной форме макроса, «арифметическими функциями»; если, например, в начале программы объявлено

*DIST(X, Y) = SQRT(X**2+ Y**2)*

то всякая последующая запись типа $DIST(e_1, e_2)$

где e_1 и e_2 – выражения типа *REAL,* будет заменена перед трансляцией (вообще говоря) на

$SQRT(e_1**2 + e_2**2)$

Имя функции и имена «формальных» параметров (здесь *DIST, X* и *Y*) могут быть объектами объявления типа.

Если исключить эту достаточно ограниченную возможность ФОРТРАНа, широко распространенные «развитые» языки редко сопровождаются системой обработки макросов. Справедливости ради надо сказать, что язык ПЛ/1 в том виде, в каком он был предложен ИБМ, имеет препроцессорную систему такого рода, которая, однако, не включена в официальный стандарт ПЛ/1.

Заслуживает интерес следующее замечание: наряду с признанием того, что ФОРТРАН – это язык более низкого уровня, чем АЛГОЛ, ПЛ/1, СИМУЛА и т.д., последние годы были расцветом препроцессоров, переводящих в базовый ФОРТРАН тексты программ, содержащих управляющие структуры, более развитые, чем в ФОРТРАНе (циклы, переключатели), и сложные управляющие структуры (в 1976 г. в США были официально учтены более 70 систем этого типа!). В качестве примера можно посоветовать указанные в библиографии статьи Кернигана и Зана.

> По–видимому, применение таких систем ставит больше проблем, чем решает; с одной стороны, пользователь обязан изучить два языка с весьма различными и даже противоречивыми свойствами, с другой – ошибка, допущенная на верхнем уровне (несоблюдение соглашений входного языка препроцессорной системы), может диагностироваться только на нижнем (при трансляции фортрановской программы, созданной препроцессором): пользователю в этом случае не остается ничего больше, как разыскивать свою ошибку, исходя из этой программы на ФОРТРАНе, которая, будучи создана системой, является, как правило, длинной и непонятной. Вместо того чтобы использовать ненастоящий ФОРТРАН и подвергаться этому виду злоключений, рекомендуется перейти к языку, предлагающему возможности корректного структурирования, или, если это реально невозможно, принять в качестве концепции программы некоторую систему обозначений высокого уровня, похожую на ту, что мы используем в этой книге, чтобы перевести затем программы на «разумный» ФОРТРАН.

IV.4.5. Передача массивов

С логической точки зрения категории аргумент, результат и модифицируемый параметр применяют к массивам точно так же, как к другим объектам, которые могут быть переданы как параметры, и мы их будем последовательно уточнять: необходимо определить «права» подпрограммы на элемент, который ей передан.

С практической точки зрения такие способы передачи, как «значение, «результат» или «значение–результат», не применимы к массивам из–за недопустимо большого времени, которое потребовалось бы на современных машинах для копирования и переписывания массивов целиком. Способ, обычно используемый в этом случае, это *передача по адресу.* Все же жаль, что обычные языки не позволяют требовать явного задания способа передачи; в АЛГОЛе W, например, спецификация *VALUE,* запрещенная для параметров–«массивов», должна была бы быть разрешенной, но означать не переписывание, а запрещение модифицировать элементы. Вместо этого массив явно передается как модифицируемый параметр, делая невозможной всякую защиту. Здесь имеет место пример слишком тесной связи между концепциями даже самых «развитых» языков программирования и проблемами физического пред-

ставления.

Таким образом, основная информация, передаваемая подпрограмме, это *адрес* первого элемента массива, являющегося фактическим параметром. В ФОРТРАНе этот адрес, действительно, представляет собой единственную информацию, которую для соблюдения стандарта языка необходимо сообщить подпрограмме.

В связи с параметрами–«массивами» ставятся и другие проблемы, отличные от проблем прав доступа:

- можно ли и нужно ли гарантировать наличие точного соответствия между числом измерений и границами в фактическом и формальном параметрах? Утвердительный ответ, дающийся в какой–то мере АЛГОЛом W и ПЛ/1, приводит к тому, что подпрограмма должна получать при каждом вызове *дескриптор массива,* содержащий, кроме адреса, сведения, описывающие число измерений и границы. В ФОРТРАНе, напротив, подпрограмма работает с формальным параметром как с именем (адресом) области памяти, которую она использует по своему усмотрению. Это решение повышает эффективность вызовов и гибкость использования, но делает призрачным любой контроль (действительно, переход за границы массива, передаваемого в качестве параметра, в ФОРТРАНе представляет источник частых и трудно обнаружимых ошибок);
- как можно передать подпрограмме некоторый *подмассив* данного массива? Например, можно ли применить программу, параметрами которой являются два одномерных массива *ВЕЩ* и которая вычисляет их векторное произведение, к строке и столбцу двух двумерных массивов, представляющих квадратные матрицы? АЛГОЛ W и ПЛ/1 дают программисту синтаксическое средство для выполнения некоторых из этих операторов. ФОРТРАН делает их возможными, давая программисту знание *способа размещения* элементов всего массива в памяти – способа, составляющего часть стандарта языка.

Методы ФОРТРАНа

В ФОРТРАНе формальный параметр «массив» представляет собой объект объявления размерности в подпрограмме. Например,

ФОРТРАН

```
      INTEGER FUNCTION MAXTAB(T)
      INTEGER
      T(100)
C     ВЫЧИСЛЕНИЕ МАКСИМУМА МАССИВА ИЗ 100
C     ЭЛЕМЕНТОВ
      MAX TAB = T(1)
      DO 100 I = 2, 100
      IF (T(I).GT.MAXTAB) MAXTAB = T(I)
100   CONTINUE
      RETURN
      END
```

Такая подпрограмма вызывается с именем массива в качестве фактического параметра; это имя объявлено также в вызывающей программе:

```
      INTEGER TAB1(100), TAB2(100) INTEGER SOMMAX
      ...
      ... инициализация TAB1 и TAB2
      ...
      SOMMAX = MAXTAB (TAB1) + MAXTAB (TAB2)
```

Более вероятно, мы пожелали бы иметь подпрограмму *MAXTAB,* умеющую оперировать с массивами произвольной размерности. На ФОРТРАНе можно написать

ФОРТРАН

```
      INTEGER
      FUNCTION MAXTAB (T,N)
      INTEGER N, T(N)
C     ВЫЧИСЛЕНИЕ МАКСИМУМА МАССИВА T
C     РАЗМЕРНОСТИ N > = 1
      MAXTAB= T(1)
      IF (N.EQ.1) GOTO 1000
            DO 100 I = 2, N
                  IF(T(I).GT.MAXTAB) MAXTAB =T(I)
100   CONTINUE
1000  RETURN
      END
```

Объявление, включающее отличные от констант границы, такие, как

INTEGER T(N)

позволено только в этом точном контексте, т.е. в подпрограмме, где таким образом объявленный массив и все отличные от констант границы, участвующие в объявлении (здесь *N*), являются **формальными параметрами.**

Важно отметить, что этой уловкой не достигается эффект «переменных» границ: в самом деле, границы массива фиксированы, они лишь уточняются вызывающей программой.

Почти во всех фортрановских системах единственной информацией для описания параметра–«массива» является адрес. Подпрограмма использует этот адрес для обращения к последовательным элементам массива. Это свойство применяют для передачи подмассивов в качестве формальных параметров. Для массива, объявленного, например, как

INTEGER TAB 3(200)

можно написать

J = MAXTAB(TAB3(100), 80)

Это означает, что *MAXTAB* применяется к 80 элементам массива *TAB3,* начиная с *TAB3(100)* (до *TAB3(179)*). Информация, передаваемая подпрограмме для описания первого параметра, это адрес 100–го элемента массива *TAB3.* Заметим, что транслятору нет необходимости обрабатывать этот случай особым образом, потому что нормальным способом передачи для переменной или элемента массива является передача по адресу.

Точно так же подпрограмме можно передавать и подмассивы более общего вида. Пусть задан массив

INTEGER T2DIM (50, 10)

Напомним (II.3.3.3), что элементы такого массива расположены «по столбцам», т.е. в порядке

T2DIM(1,1), T2DIM(2,1), ..., T2DIM(50,1), T2DIM (1,2), ...,

T2DIM (50,10)

Вызов

M = MAXTAB (T2DIM (1,1), 50)

разрешен: он будет присваивать переменной *M* значение максимума из 50 элементов массива *T2DIM*, начиная с *T2DIM(1,1)*, т.е. с первого столбца *T2DIM*;

T2DIM(1,1), T2DIM (2,1),..., T2DIM(49,1), T2D1M(50,1)

В более общем виде, для $1 \le I \le 10$, *MAXTAB(T2DIM(1,1), 50)* имеет своим значением максимум *I*–го столбца массива *T2DIM*. Примеры более общих подмассивов:

MAXTAB (T2DIM(1, 1), 100) применяется к первым двум столбцам;

MAXTAB(T2D1M(26, 1), 100) применяется ко второй половине первого столбца, второму столбцу и первой половине третьего.

Таким же образом можно передать подпрограмме все множество смежно расположенных элементов. Напротив, вычисление максимума элементов строки в помощью *MAXTAB* невозможно.

В ФОРТРАНе подпрограмме могут передаваться только подмассивы, элементы которых расположены в смежных областях памяти.

В качестве примера следующая подпрограмма могла бы быть полезной для некоторых задач исследования операций:

ФОРТРАН

```
      INTEGER FUNCTION MINMAX(MAT, M,N)
      INTEGER M,N, MAT(M,N), MAXCOL
C     ВЫЧИСЛЕНИЕ МИНИМУМА СРЕДИ МАКСИМУМОВ
C     СТОЛБЦОВ МАТРИЦЫ MAT
C     ПРЕДПОЛАГАЕТСЯ, ЧТО M >= 1, N > = 1
      MINMAX = MAXTAB (MAT (1,1), (M)
      IF (N.EQ.1) GOTO 1000
          DO 100 I = 2,N
              MAXCOL = MAXTAB (MAT(1,1), M)
              IF (MAXCOL .LT. MINMAX)
              MINMAX = MAXCOL
100   CONTINUE
1000  RETURN
      END
```

Сравнивая *MAXTAB* и *MINMAX*, можно заметить, что в первой программе величина *N*, которая фигурирует в объявлении размера массива *T*, совершенно формальна, если система ФОРТРАНа не проверяет правильность индексов массива; в этом случае можно было бы заменить *N* на некоторый параметр или произвольную константу, не влияя на эффект действия подпрограммы.

В *MINMAX*, напротив, объявление

INTEGER MAT(M, N)

задает важную величину *M*, позволяющую системе вычислить адрес, соответствующий любому элементу массива *MAT*. Точнее, в силу размещения «по столбцам» соответствующий адрес в *MAT (I, J)* для $1 \le I \le M$ и $1 \le J \le N$ вычисляется по формуле

адрес первого элемента из *MAT* + $(M \times (J - 1) + I - 1)$t

где t – размер области, выделенной одному целому (измеренный в словах, в байтах и т.п.).

В заключение можно сказать: способ, которым ФОРТРАН обрабатывает

формальные параметры–«массивы», предлагает достаточно большую гибкость применений. Передача подмассива возможна при условии, что она предусмотрена: массив организован в виде подмассивов, которые могут обрабатываться отдельно. Наконец, поскольку передается только адрес, возможности контроля очень ограничены и ошибки индексации в подпрограмме, например, зачастую трудно обнаруживаемы.

Методы ПЛ/1

В ПЛ/1 параметр–массив должен объявляться обычно в подпрограмме

ПЛ/1

```
PROGR: PROCEDURE OPTIONS (MAIN);
        DECLARE TAB1(100) DECIMAL FIXED;
        MAXTAB: PROCEDURE (T) RETURNS (DECIMAL FIXED)
                DECLARE T(100) DECIMAL FIXED,
                        MAX DECIMAL FIXED;
                /*ВЫЧИСЛЕНИЕ МАКСИМУМА ИЗ Т*/
                MAX = T(1);
                DO 1 = 2 TO 100;
                    IF T(I) MAX THEN MAX = T(I);
                    END;
                RETURN (MAX);
                END MAX TAB;
          ... инициализация TAB1 ...
          PUT LIST (MAXTAB(TABl));
          END PROGR;
```

Если необходимо предусмотреть, чтобы некоторые границы были не определены, в объявлении формального параметра–массива заменяют соответствующие спецификации размера на звездочки. Примеры:

```
MAXTAB: PROCEDURE (T, M, N) RETURNS (DECIMAL FIXED);
        DECLARE T(*) DECIMAL FIXED,M DECIMAL FIXED
        N DECIMAL FIXED, MAX DECIMAL FIXED;
        /* МАКСИМУМ ОДНОМЕРНОГО МАССИВА Т
          С ГРАНИЦАМИ М И N*/
        ... /* ПРЕДЫДУЩИЙ АЛГОРИТМ, В КОТОРОМ
            1 ЗАМЕНЕНА М, А 100 – N *:/ ...
PRODMAT: PROCEDURE (A, B, C, N);
            DECLARE (A (*,*),B(*,*),C(*,*)) DECIMAL FLOAT,
                    N DECIMAL FIXED;
            /* ПРИСВАИВАНИЕ А РЕЗУЛЬТАТА МАТГИЧ–
            НОГО ПРОИЗВЕДЕНИЯ ВИС; ПРЕДПОЛАГА–
            ЕТСЯ, ЧТО ВСЕ ТРИ МАТРИЦЫ ИМЕЮТ РАЗ–
            МЕРЫ (1:N, 1:N) */
            ...
            DECLARE (MAT1(10, 10), MAT2(10, 10),
                    MAT3(10, 10)) DECIMAL FLOAT;
            ...инициализация MAT2 и MAT3...
            CALL PRODMAT (MAT1,MAT2,MAT3,10);
            ...
```

Подмассив можно в той же степени использовать как фактический параметр. Пусть, например, есть массив *T*

DECLARE T(10, 20:30, 5) ...

Тогда можно передать *T(*, 25, 3)* подпрограмме, оперирующей с одномерными 10–элементными массивами: передаваемый массив содержит элементы

T(1,25,3) T(2,25,3) ... T(10,25,3)

так же, как фактический параметр вида

T(, 27, *)*

означает подмассив, сформированный из элементов

T(1,27,1) T(1,27,2) ... T(1,27,5)
T(2,27,1) T(2,27,2) ... T(2,27,5)
7(10,27,5)

Возможности построения подмассивов еще более расширены путем применения псевдоиндексов *iSUB* с которыми мы познакомимся в гл. V (V.9.5).

Методы АЛГОЛа W

Обработка параметров–«массивов» в **АЛГОЛе W** достаточно близка к методам ПЛ/1. Главное различие состоит в том, что обозначение звездочкой является единственным допускаемым для объявлений размерности этого типа параметров в подпрограммах; запрещено указывать их границы, которые для фактических параметров фиксируются вызывающей программой

АЛГОЛ W

```
        BEGIN
        INTEGER ARRAY TAB1(–5::50);
        INTEGER ARRAY TAB2 (0::1000);
        INTEGER ARRAY DTAB(–10::10, 0::15);
        INTEGER PROCEDURE MAXTAB (INTEGER ARRAY T(*),
                INTEGER VALUE BORNE_INF, BORNE_SUP);
                COMMENT: МАКСИМУМ ОДНОМЕРНОГО
                              МАССИВА С ГРАНИЦАМИ
                              BORNE, INF И BORNE_SUP;
                BEGIN INTEGER MAX;
                MAX : = –MAXINTEGER
                COMMENT: MAXINTEGER – НАИБОЛЬШЕЕ
                                      ЦЕЛОЕ, ОПРЕДЕ–
                                      ЛЕННОЕ В МАШИ–
                                      НЕ;;
        FOR I: = BORNE  INF UNTIL BORNE  SUP DO
                IF T(I) > MAX THEN MAX : = T(I);
        MAX
        END MAX TAB;

        ... инициализация TAB1, TAB2, DTAB ...;
(1)     WRITE (MAXTAB(TAB1, –5,50));
(2)     WRITE (MAXTAB(TAB2, 100., 500));
(3)     WRITE (MAXTAB(DTAB(*,7, –10, 10))
        END.
```

Строка, помеченная (1), это применение *MAXTAB* к *TAB1*. Строка (2) применяет

MAXTAB к подмассиву *TAB2,* образованному из элементов с индексами от 100 до 500. Наконец, строка (3) применяет *MAXTAB* к 7–му «столбцу» *DTAB,* т.е. к элементам

DTAB (–10,7) DTAB (–9,7) ...DTAB (10,7)

В более общем виде в АЛГОЛе W можно передать подмассивы, соответствующие произвольным «блокам» (или «клеткам») матрицы. Символ «звездочка» в объявлении формального параметра–«массива» описывает число измерений, а не границы:

```
PROCEDURE PRODUIT_MATRICTEL (LONG REAL ARRAY A, B, C(*,*);
                             INTEGER VALUE N);
COMMENT: PRODUIT MATRICIEL – МАТРИЧНОЕ ПРОИЗВЕДЕНИЕ,
         ПРИСВАИВАНИЕ А ЗНАЧЕНИЯ МАТРИЧНОГО
         ПРОИЗВЕДЕНИЯ В И С.
         ВСЕ ТРИ МАТРИЦЫ ПРЕДПОЛАГАЮТСЯ ИМЕЮЩИМИ
         РАЗМЕРНОСТЬ (1::N, 1 : : N )
```

Практически почти всегда необходимо включать параметры, обозначающие соответствующие границы или, по крайней мере часть из них, если они, как в нашем примере, не задаются неявно.

Заключение

Методы, предлагаемые АЛГОЛом W и ПЛ/1, позволяют программисту не заниматься физическим размещением элементов в памяти, предоставляя ему синтаксические средства для. обозначения подмассива. С точки зрения системы информация, передаваемая при каждом вызове, более сложна, чем в ФОРТРАНе: это «дескриптор», содержащий, кроме адреса, информацию (число измерений, границы, шаг изменения индексов), обеспечивающую доступ к произвольному элементу фактического параметра по формальному параметру и списку индексов.

В этих языках программист меньше связан конкретным способом расположения элементов в памяти. Платить за это приходится ценой потери гибкости подпрограмм; так, в АЛГОЛе W невозможно применить *MAXTAB* к двумерному массиву, зато передаваемая информация богаче и система лучше управляет совместимостью параметров и законченностью их обработки в программе.

В нашей алгоритмической нотации мы можем работать только с подмассивами, соответствующими смежным подмассивам индексов («клеткам»). Так, если t и u объявлены как

```
массивы t [0:50] : ЦЕЛ,
        u[–1:12, 1:30] : ЛОГ
```

то допустимыми подмассивами массивов t и u будут

```
t[i : j]
```

и

```
u[a : b, c : d]
```

где i, j, a, b, c, d – целые, такие, что $0 \le i \le j \le 50$, $-1 \le a \le b \le 12$ и $1 \le c \le d \le 30$.

IV.4.6. Передача подпрограмм

Мы видели, что формальный параметр может соответствовать подпрограмме. Так, подпрограмма численного интегрирования может иметь вид

```
программа интеграл: ВЕЩ (аргументы a, b: ВЕЩ, f:(ВЕЩ → ВЕЩ))
    | приближенное вычисление ∫_b^a f(x)dx
    | ...
```

С ее помощью можно вычислить, например, интеграл (0.5, 0.7, SINUS).

Метод передачи, применяемый к этому типу параметров, это **передача по адресу:** вызывающая программа передает подпрограмме адрес соответствующего кода фактического параметра (SINUS в вышеприведенном примере).

Главная проблема, которую ставит этот тип параметров, это синтаксическая проверка (транслятором) соответствия между числом фактических параметров и их типами, с одной стороны, и спецификациями соответствующих формальных параметров – с другой; важно не разрешать, например, вызовы вида

интеграл (0.5,0.7, imprimcarac)

где imprimcarac – это подпрограмма (ЛИТ × ЦЕЛ → ПУСТО).

Из наших трех языков только ПЛ/1 разрешает задавать транслятору необходимые управляющие атрибуты. Формальный параметр может быть объявлен с атрибутом

ENTRY (mun1, mun2, ..., munn)

который указывает, что речь идет о подпрограмме, и уточняет тип ее параметров; атрибут *RETURNS(mun)* уточняет, задается ли тип результата. Например,

```
INTEGR PROCEDURE(A,B,FONCTION) RETURNS(BINARY FLOAT(53));
  DECLARE (A, B) BINARY FLOAT (53),
          FONCTION ENTRY (BINARY FLOAT (53))
                   RETURNS (BINARY FLOAT (53));
  ... алгоритм, вычисляющий интеграл от ФУНКЦИИ между A и
                                                  B ...;
END INTEGER;
```

Объявление формального параметра *ENTRY* необязательно (к сожалению), если контекст позволяет его заменить, например если параметр встречается в операторе *CALL*. Можно также (опять же, к сожалению) не уточнять тип всех параметров.

Соответствующий фактический параметр должен быть процедурой совместимого типа. Можно также указывать для нее *ENTRY ...RETURNS ...,* если возможна двусмысленность.

Если фактический параметр является именем процедуры без параметров, он воспринимается имеющим атрибут *ENTRY;* напротив, если он заключен в скобки, будет иметь место передача значением (см. IV.4.2): передается результат выполнения процедуры, а не имя этой процедуры.

ФОРТРАН и АЛГОЛ W более примитивны в этом отношении. В ФОРТРАНе формальный параметр–«подпрограмма» не объявляется, если не считать объявлением тип подпрограммы–«выражения» (*FUNCTION*)*:*

```
DOUBLE PRECISION FUNCTION INTEGR (A, B, FONC)
DOUBLE PRECISION A, B, FONC
```

FONC в этом случае распознается как подпрограмма, только если *INTEGR* использует ее в выражении типа *FONC(x),* которое обозначает, что *FONC* есть подпрограмма *FUNCTION,* поскольку она не объявлена как массив. Подпрограмма *SUBROUTINE* указывается, как таковая, тем, что она встречается в операторе *CALL.*

Однако, во всяком случае, соответствующий фактический параметр должен быть объявлен в любой вызывающей программе, имеющей тип *EXTERNAL,* т.е. подпрограммой. Это справедливо, даже если речь идет об известных в ФОРТРАНе «стандартных» процедурах, таких, как *DSIN,* которая дает синус с удвоенной точностью:

```
EXTERNAL DSIN
DOUBLE PRECISION DSIN, I
```

```
I = INTEGER (0.0D0, 1.0D0, DSIN)
```

В АЛГОЛе W формальный параметр–«подпрограмма» обозначается просто с помощью *PROCEDURE* или *тип PROCEDURE*. Например,

```
LONG REAL PROCEDURE INTEGRALE
        (LONG REAL VALUE A, B,
        LONG REAL PROCEDURE FONCTION);
```

Вызов тогда может иметь вид

```
WRITE (INTEGRALE (0,1, LONGSIN))
```

IV.4.7. Резюме

Таблицы на Рис. IV.7 и Рис. IV.8 показывают свойства различных рассмотренных здесь способов передачи и их отношения с ФОРТРАНОМ, ПЛ/1, АЛГОЛом W.

Способ передачи	Значение	Результат	Значение результат	Адрес	Имя
Обозначение	формальный ← фактический на входе в подпрограм му	формальный ← фактический на выходе из подпрограммы	формальный ← фактический на входе в подпрограмму; фактический ← формальный на выходе из подпрограммы	передача адреса	текстуальная подстановка после исключения возможных конфликтов имен
Фактический параметр – константа	да			да (если подпрограмма не меняет параметр)	да (если подпрограмма не меняет параметр)
Фактический параметр – выражение	да			да (если подпрограмма не меняет параметр)	да (если подпрограмма не меняет параметр)
Фактический параметр – переменная или элемент	да	да	да	да	да°
Параметр – массив или подмассив	практически не используется	практически не используется	практически не используется	да	да
Параметр – подпрограмма				да	да

Рис. IV.7 Типы параметров и способы передачи: определение, возможности и несовместимости.

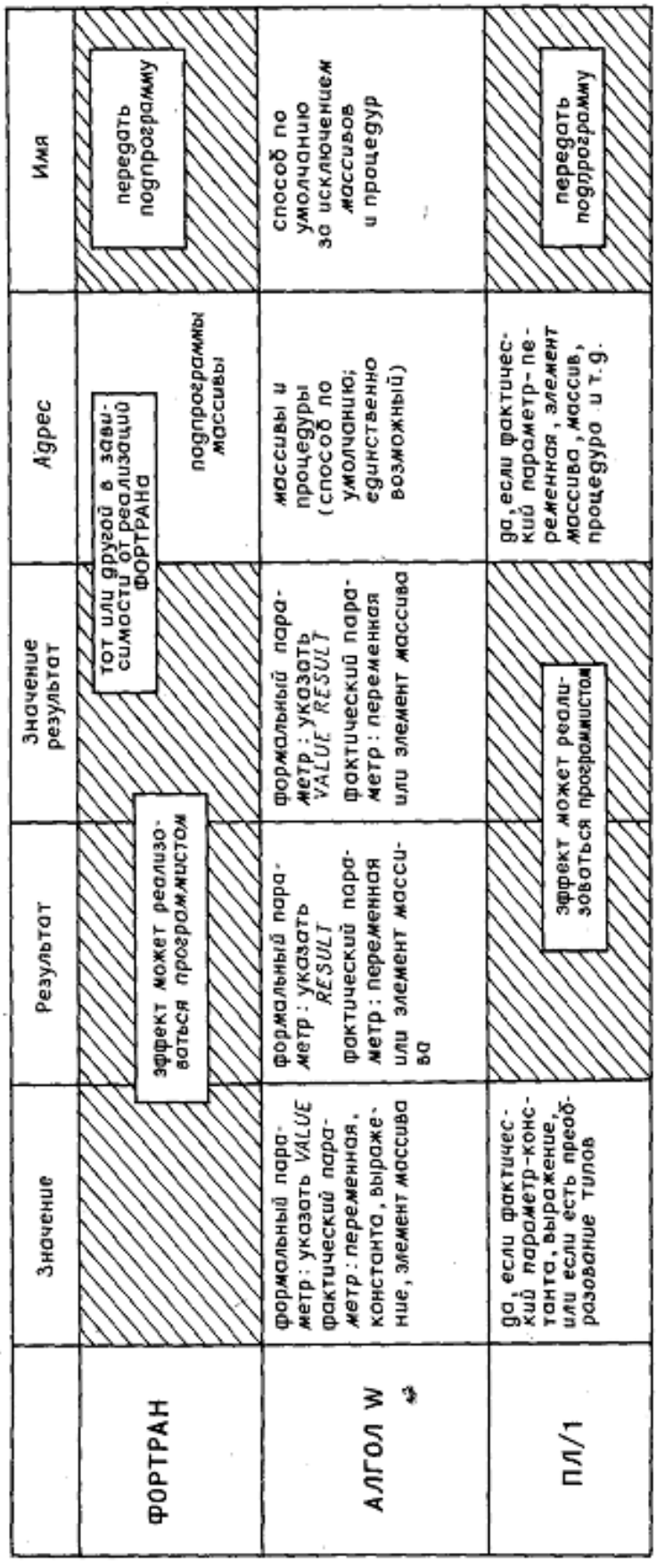


Рис. IV.8 Способ передачи аргументов в ФОРТРАНе, АЛГОЛе W, ПЛ/1 ("способ по умолчанию" означает, что речь идет о таком случае, когда программист не уточняет никакого конкретного способа).

IV.5. Обобществление данных

IV.5.1. Определение

Предыдущий раздел был посвящен методам, которые позволяют программе явно передавать информацию в момент вызова. Возможен также доступ различных программных модулей к общим данным без того, чтобы передавать их в качестве параметров. Это называют *обобществлением данных.* В этом разделе рассматривается практический способ реализации такого обобществления; затем сравниваются достоинства передачи параметров и обобществления данных.

Замечание

Всюду в этой главе вызовы подпрограмм имеют «иерархическую» структуру, благодаря которой в каждый момент только одна программа является активной: вызов подпрограммы автоматически выключает управление вызывающей программы до конца выполнения вызванной подпрограммы. Это позволяет не рассматривать возможность конфликтов доступа, порождаемых обобществлением данных. Эта проблема становится, напротив, ключевой, когда рассматриваются *параллельные процессы;* по этому поводу полезно обратиться к [Крокус 75] или [Бринч Хансен 73].

IV.5.2. Языки блочной структуры

«Общий доступ к данным» означает, что подпрограмма может работать с переменными, элементами массивов и т.д., которые не передаются явно как параметры. В АЛГОЛе W и ПЛ/1 стандартные правила структуры блока (III.5.1) позволяют всякой подпрограмме иметь доступ к любой переменной или любому массиву, объявленному в блоке («группе *BEGIN*» или «группе *PROCEDURE*» в ПЛ/1), где объявлена подпрограмма, или в любом включающем блоке, если только тот же идентификатор не объявлен заново в подпрограмме. Рис. IV.9 показывает схему, соответствующую этой ситуации.

```
переменные  a, b : ВЕЩ;
массив      x [1:100] : СТРОКА;
программа   p : ЦЕЛ (параметры ...; результаты...)
                ...
                {p с доступом к a, b, x}
                ...
программа q (параметры...; результаты...)
      переменная y: СТРОКА;
      программа r (параметры...)
      {r с доступом к y, к аргументам q, к a, b, x}
      {q с доступом к a, b, x}
      ...
      ...
.... {основная программа}
```

Рис. IV.9 Обобществление с помощью применения правил структуры блока.

IV.5.3. Методы ФОРТРАНа: понятие общей области

ФОРТРАН предлагает оригинальный метод обобществления данных – общие области, так называемые области *COMMON*. «Общая область» в ФОРТРАНе – это

«модуль данных», аналогичный программным модулям: он позволяет «озаглавить» единым именем набор различных объектов (переменные массивы).

Объявление общей области имеет вид

COMMON/имя–области/V 1, V2, ..., Vn

где *имя–области* есть идентификатор, который обозначает общую область, а *V1, V2, ..., Vn* – имена переменных или массивов. Это объявление должно присутствовать во всех программных модулях, которые используют эту общую область, наряду с возможными объявлениями типов, связанных с объектами *V1, ..., Vn*. Если один из этих объектов есть массив, то указания размерности и границ могут составить часть объявления общей области или фигурировать отдельно. Например,

```
COMMON /CP1/ PILE, SOMPIL
INTEGER PILE (500), SOMPIL
```

можно также записать в виде

```
COMMON /CP1/ PILE (500), SOMPIL
INTEGER PILE, SOMPIL
```

Имя общей области (здесь *CP1*) не имеет никакого смысла само по себе и не является, в частности, именем переменной. Его единственная роль – позволить различным программным модулям иметь доступ к одним и тем же объектам, обозначив их глобально одним идентификатором.

Важно подчеркнуть, что имена, выбранные для этих объектов, напротив, имеют только локальный смысл: если в подпрограмме *SP1* имеются строки

```
SUBROUTINE SP1 (  )
COMMON /CP1/ PILE (500), SOMPIL INTEGER PILE, SOMPIL
```

а в программе *SP2* – строки

```
SUBROUTINE SP2 (  );
COMMON /CP1/ P(500), S INTEGER P, S
```

то *P* и *PILE, S* и *SOMPIL* означают соответственно одни и те же объекты; если *SP2* выполняет

```
P(1) = 30
```

то первый элемент массива *PILE* будет равен 30 к следующему выполнению *SP1*.

Таким образом, только имя общей области, а не имена ее элементов обеспечивает связь между этими элементами в различных программных модулях (ср. соответствие «имена фактических параметров – имена формальных параметров»). Тем не менее очень рекомендуется *обозначать эти объекты одним и тем же именем во всех программных модулях;* это, с одной стороны, увеличивает ясность программы, а с другой стороны, не видно, какие преимущества можно извлечь, давая разные имена одним и тем же объектам (кроме редких особых случаев).

Важно отметить, однако, что с помощью одного имени общей области можно связать различные объекты. С этой точки зрения ФОРТРАН является почти «машинным языком» в том смысле, что для него единственно важным является отношение смежности в памяти. В той же мере, как можно передать три соседних столбца массива (30, 50) подпрограмме, обрабатывающей массив из 90 элементов, точно так же можно иметь в двух разных подпрограммах соответствующие объявления:

```
      COMMON /CC/ ITAB(100)
и     COMMON /CC/ IT1(50), IT2(48), I, J
```

(предполагая, что все рассматриваемые элементы имеют тип *INTEGER).* В этом случае *IT1(40)* означает тот же элемент, что и *ITAB(40); IT2(27)* – тот же элемент,

что *ITAB(77); I* – тот же элемент, что и *ITAB(99)*, и т.д. Этот вид объявлений увеличивает вероятность необнаруживаемых ошибок: даже если система ФОРТРАНа не проверяет систематически индексы элементов используемых массивов, ошибочное обращение типа *ITAB(K)* при *K,* равном 101, может в некоторых ситуациях обнаружиться, потому что оно порождает неразрешенную ссылку на защищенную область памяти; напротив, *IT1(L)* при *L*, равном 51, порождает совершенно законный доступ к *ITAB(51)* (т.е. *IT2(1)*) и соответствующую ошибку, будет трудно найти. Предоставляем читателю угадать наши чувства по отношению к манипуляциям такого сорта. Некоторые системы идут еще дальше: допускается взаимное соответствие элементов различных типов при условии, что общий объем памяти остается тем же: например, на машинах ИБМ 360/370 величина *DOUBLE PRECISION* = одно *REAL* + одно *INTEGER* = *2* слова. Мы не будем этого требовать. Ниже показано «хорошее» использование *COMMON* в ФОРТРАНе.

Техническая деталь: некоторые системы требуют из соображений ограничений размещения в памяти («выравнивания строк»), чтобы самые длинные элементы области *COMMON* размещались в начале области. Например, на ИБМ 360/370 объявление

COMMON /CCC/ DOUB, ENT, L1

где *DOUB* имеет тип *DOUBLE PRECISION* (8 байт), *ENT* тип *INTEGER* (4 байт), а *L1* – тип *LOGICAL*1* (1 байт; тип, характерный для этих машин), должно содержать эти объекты в указанном порядке.

Заметим также, что если объекты, представленные в общей области, могут быть инициализированы присваиванием в любом программном модуле, содержащем объявление общей области, то они могут фигурировать в предписании *DATA* только в «псевдопрограмме», специально предназначенной для такого действия; такая псевдопрограмма, называемая *BLOCK DATA,* имеет следующий вид:

```
BLOCK DATA
      ... {последовательность объявлений COMMON,
           объявлений типов и директив DATA,
           за исключением любых выполняемых операторов...}
END
```

Отметим для памяти существование специальной общей области без имени (или с пустым именем), называемой «областью пробелов». Ее объявление записывается в виде

COMMON V_1 V_2, ... V_n

Эта общая область играет особую роль в истории ФОРТРАНа, потому что некоторые системы (такие, как CDC 6600/7600) назначают этой области все выделенное данной работе место в памяти, не занятое программами и данными. В этом случае область пробелов может управляться как «куча» *(«heap»)* в том смысле, в котором это слово будет введено в V.2, т.е. область, где могут развертываться структуры данных, размера, не предусматривавшегося перед выполнением: стеки, списки, деревья и т.д. К сожалению, это не общее свойство всех систем, и оно приводит к некоторым неприятностям, если пытаться извлечь из него слишком много преимуществ.

IV.5.4. Обсуждение: обобществление или передача?

Использование обобществленной переменной, вообще говоря, эквивалентно передаче параметра по адресу или как «значение–результат». Естественно ставится вопрос: в каком случае обобществление оказывается предпочтительнее передачи параметра? Ниже обсуждаются несколько оснований для такого выбора.

В пользу обобществления переменных могут быть высказаны следующие предложения:

- простота записи: можно не писать объявления формальных параметров; для ФОРТРАНа это не так (*COMMON);*
- выигрыш в эффективности: можно уйти от копирования значений и

переписывания результатов (передачи значения–результата) или от пересылок адресов (передача по адресу); действительно, адреса обобществляемых данных определяются раз и навсегда в момент размещения программы в памяти (загрузка или редактирование связей). Следует, однако, заметить, что этот выигрыш эффективности значителен только в случае коротких и очень часто выполняемых подпрограмм. Он исчезает в операционных системах с виртуальной памятью;

- в случае языков, не имеющих блочной структуры, каким является, например, ФОРТРАН, часто возникают ситуации, в которых несколько подпрограмм вместе управляют некоторым количеством переменных; речь может идти, например, о подпрограммах доступа к структуре данных (стек, очередь и т.д.; см. гл. V) и управляющих переменных этой структуры (указатель верхушки стека и т.д.). Передавать эти переменные как параметры при каждом вызове одной из подпрограмм и утомительно и опасно, так как часто бывает нужно «маскировать» их от вызывающих программ, которым не известна внутренняя структура подпрограмм. В таком случае оправдывается использование *COMMON,* группирующего эти переменные. Одно из неудобств этого метода: ничто не гарантирует исключительности применения этого *COMMON* в подпрограммах, имеющих на это «естественное» право.

Следует резервировать использование обобществленных данных для тех фиксированных случаев, когда несколько программ обрабатывают группу данных, действительно общих для них в том смысле, что нет оснований полагать, будто эти данные принадлежат скорее одной из этих подпрограмм, чем другой. Во всех других случаях, когда имеет место пересылка данных, принадлежащих одной программе, рекомендуется передача с помощью параметров.

Мы отвергаем в связи с этим один из доводов, иногда приводимых в пользу применения обобществляемых переменных, представляющий собой убеждение в том, что это единственное жизнеспособное решение, когда объем информации, которой обмениваются программы, становится большим. Если речь идет о попытках передать слишком большое количество информации между двумя программными модулями, это, на наш взгляд, свидетельствует просто о плохом замысле программы либо о плохом разделении ее на модули, и такую программу следует переделать.

Недавние работы по поводу построения больших программ (см., например, [Парнас 72]) со всей определенностью настаивают на необходимости *ограничивать доступность информации.* Надежность сложной системы в большой степени связана с «узостью» и ограниченностью «интерфейсов» между ее различными модулями.

Фортрановский способ общения, в котором интерфейсы явно определены с помощью объявлений *COMMON,* с этой точки зрения имеет предпочтение перед структурой АЛГОЛа, где любая процедура может иметь доступ к любому объекту во включающем блоке. В последних «алголоподобных» языках много усилий сделано, чтобы ограничить *права доступа программ* указанием способов «защиты», связанных с каждым объектом. В этой области сходятся интересы специалистов по языкам программирования и специалистов по операционным системам: если последние мало–помалу научились обрабатывать объекты (машинные представления, ресурсы, процессы, ...), которые они рассматривают как обычные программистские вещи со всеми возникающими в связи с этим задачами (объявления, передачи параметров, структуры связанных данных...), то первые извлекли выгоды из опыта, приобретенного в управлении обобществленными объектами, защитой, в определении прав доступа и т.д.

Вернемся к практическим заботам: из всего предшествующего должно быть ясно, что такое понятие, как общая область, может употребляться удачно или неудачно.

Области *COMMON* в ФОРТРАНе дали основание для злоупотреблений: некоторые программисты систематически включают в начало каждого программного модуля гигантское объявление области *COMMON,* содержащей до нескольких сотен элементов и отвечающей на какое–нибудь выразительное имя, например, *ЧУШЬ.* В этом случае, конечно, нечего заниматься информацией, передаваемой каждой подпрограмме: все имеют доступ ко всему! Само собой разумеется, что любое управление в этом случае невозможно и что ошибка, допущенная подпрограммой, например, в работе с индексами, может иметь далеко идущие последствия.

Мы будем требовать, напротив, чтобы *COMMON* группировал небольшое число связанных между собой данных и общих для небольшого числа программных модулей. Можно сказать, ссылаясь на определение разд. I.7 и игнорируя различия физического представления: *область COMMON – это маленький файл.*

IV.6. Подпрограммы и управление памятью

IV.6.1. Свободные переменные; массивы с фиксированными при выполнении границами

Каждая подпрограмма работает с некоторым числом принадлежащих ей объектов. Так, мы видели, что для того, чтобы иметь доступ к входным и выходным параметрам, программа должна обладать локальными копиями этих параметров (передача значением, передача результата, передача значения–результата), либо их адресами (передача по адресу), либо процедурой доступа (передача именем). Кроме этих «аргументов» и «результатов», подпрограмма использует, вообще говоря, некоторое число принадлежащих ей «локальных переменных».

Задача управления памятью ставится, следовательно, так, чтобы предложить каждой подпрограмме независимую область. Предлагаемый метод зависит, в частности, от ответов на два следующих вопроса:

а) Сохраняет ли локальная переменная подпрограммы свое последнее значение к следующему вызову? Переменную, обладающую таким свойством, называют *свободной переменной*[1].

б) Можно ли строить и использовать массивы, размеры которых не фиксированы при выполнении?

Свободные переменные необходимы в некоторых ситуациях. Один из наиболее типичных случаев предполагает необходимость подсчета числа вызовов подпрограммы; для этого используют переменную–«счетчик», значение которой увеличивается на единицу при каждом вызове. Ясно, что эта переменная принципиально принадлежит подпрограмме, а не одной из программ, которые могут вызывать эту подпрограмму; для того чтобы такую переменную можно было объявить локальной, нужно, чтобы она была свободной.

Что касается **массивов, размеры которых могут быть изменены** перед каждым выполнением подпрограммы, они полезны, когда подпрограмма использует массив, размер которого зависит, например, от параметра, и надо избежать потерь памяти, которая может быть вызвана систематическим резервированием памяти максимально возможного размера в предположении, что он известен.

Свободные переменные ставят серьезную проблему *инициализации.* Поскольку

---

[1] При переводе использован этот введенный АЛГОЛОМ и сложившийся уже термин; авторы называют такие переменные *«остаточными»* (remanentes). – *Прим. перев.*

они являются локальными в подпрограмме, только эта подпрограмма и может присвоить им значения; но только «вызывающие» программы могут принимать решения о задаваемых им начальных значениях! В простейшем случае начальное значение всегда одно и то же (например, 0 для обсуждавшегося выше счетчика), и нет необходимости вновь инициализировать его в ходе выполнения программы. Тогда можно инициализировать переменную «статически», т.е. до выполнения, при трансляции или загрузке с помощью директивы типа *DC* (объявление константы) в ассемблере, *DATA* в ФОРТРАНе или атрибута *INITIAL* в ПЛ/1 (такая проблема не ставится в АЛГОЛе W, который не имеет свободных переменных). В гл.II мы говорили, что директива *DATA* и атрибут *INITIAL* *должны* использоваться для определения символических констант, а не для инициализации переменных: инициализация свободных переменных – это исключение.

В тех случаях, когда начальное значение зависит от вызывающей программы, есть два не очень элегантных средства. Первое из них состоит в выделении подпрограмме параметра типа ЛОГ (назовем его первый–вызов); тело подпрограммы тогда содержит оператор

если первый–вызов **то**
| инициализация свободных переменных

и вызывающая программа поставляет фактический параметр, равный истине при первом вызове и лжи при каждом следующем (см. неумелую попытку применения этого метода в упражнении III.3). Второй способ состоит в том, чтобы сделать свободную переменную доступной извне (включая ее в ФОРТРАНе в область *COMMON* и объявляя ее во включающем блоке в языках блочной структуры); тогда можно написать специальную подпрограмму инициализации.

Действительно, чтобы получить удовлетворительное решение проблемы инициализации свободных переменных, надо выйти за рамки ФОРТРАНа, АЛГОЛа W и ПЛ/1 и вернуться, например, к СИМУЛе 67. Мы затронем этот язык в нескольких словах ниже в IV.7.

IV.6.2. Статическое и динамическое распределения

Два фундаментальных метода управления памятью, принадлежащей программному модулю, называются *статическое распределение* и *динамическое распределение.*

- при **статическом распределении** область памяти, выделяемая каждой подпрограмме, фиксирована и определена перед первым выполнением (при трансляции или загрузке) в зависимости только от текста программы;
- при **динамическом распределении** область, принадлежащая подпрограмме, напротив, определяется заново перед каждым из ее выполнений в зависимости от потребностей, уточняемых в момент вызова.

Ответ на два поставленных выше основных вопроса, таким образом, очевиден в случае статического распределения и динамического распределения:

- *при статическом распределении* одна и та же область памяти всегда связывается с одной и той же подпрограммой; переменные и массивы естественно являются свободными, если области, выделенные различным подпрограммам, отделены друг от друга. И наоборот, при фиксированном размере этих областей массивы (как и другие структуры данных) должны иметь размер, определенный раз и навсегда при трансляции;
- *при динамическом распределении* невозможно гарантировать, чтобы переменная

сохраняла свое последнее значение от одного вызова до другого, поскольку область памяти, связанная с подпрограммой, не обязательно та же самая.

Напротив, можно ожидать, что последовательные выполнения подпрограммы могут каждый раз заново фиксировать размер массивов и других структур данных при условии, что элементы, участвующие в вычислении этого размера (т.е. границы для массивов), известны перед каждым выполнением.

Какова сравнительная эффективность этих методов? Нужно различать проблемы времени и пространства:

- в использовании **пространства** памяти динамическое распределение, очевидно, более эффективно, потому что оно позволяет резервировать для каждой подпрограммы при каждом из ее выполнений только такой объем памяти, который действительно необходим для ее выполнения, а не максимально возможный размер;
- с точки зрения **времени** вычислений ответ менее очевиден. На машинах классического типа, названных «фон–неймановскими» в гл. I, где память представляет множество N последовательно расположенных слов, совершенно эквивалентных с точки зрения доступа, статическое распределение, вообще говоря, более эффективно: при этом методе *связь* между именами переменных и областями памяти устанавливается только один раз, тогда как при динамическом распределении необходимо для каждого нового выполнения подпрограммы найти и резервировать свободную и достаточно большую область, что требует времени центрального процессора в ущерб непосредственным вычислениям. Следует, однако, заметить, что иерархическая структура вызовов подпрограмм в обычных языках позволяет использовать для управления памятью очень простую структуру данных – *стек,* как это будет видно в следующей главе. Поэтому архитектура некоторых машин (с использованием базовых регистров) может пренебречь потерей эффективности по сравнению со статическим распределением. Архитектура других машин, существенно отличающихся от наиболее классических моделей («стековые» машины: серия Burroughs B570O/B670O; ICL 2900), была специально задумана, чтобы позволить эффективное использование динамического распределения.

Наши три языка дают хорошую иллюстрацию понятий статического и динамического распределений, поскольку АЛГОЛ W использует исключительно второе, трансляторы ФОРТРАНа – почти всегда первое, а ПЛ/1 – и то и другое.

IV.6.3. ФОРТРАН

В ФОРТРАНе все было предусмотрено для того, чтобы распределение памяти могло быть статическим; это самый простой метод, реализуемый на классических машинах. Он фактически употребляется в подавляющем большинстве трансляторов, хотя и не провозглашается явно стандартом языка.

Из этого непосредственно вытекает, что массивы в ФОРТРАНе имеют фиксированные границы, определенные при трансляции, и что переменные и массивы являются свободными почти во всех системах ФОРТРАНа.

Ниже приведен пример программы, использующей свободную переменную в ФОРТРАНе. Программа печатает на строке десять целых элементов массива, поставляемого в качестве параметра, переходя на следующую страницу всякий раз, когда напечатано 50 строк. Следовательно, надо ввести счетчик; соответствующая переменная, названная здесь *COMPT,* инициализируется директивой *DATA*.

ФОРТРАН

```
      SUBROUTINE IMP50 (TABENT)
      INTEGER TABENT (10)
C     НАПЕЧАТАТЬ ЭЛЕМЕНТЫ МАССИВА TABENT НА
C     СТРОКЕ, ПЕРЕХОДЯ НА НОВУЮ СТРАНИЦУ ПОСЛЕ
C     50 СТРОК
      INTEGER COMPT
      DATA COMPT /0/
C
C       ПЕЧАТЬ
      PRINT 1Q000. TAB
10000   FORMAT (IX, 10112)
      COMPT = COMPT + 1
      IF (COMPT. LT. 50) GOTO 100
            COMPT = 0
C     СЛЕДУЮЩИЙ ОПЕРАТОР ПЕРЕВОДИТ СТРАНИЦУ
            PRINT 10001
10001                 FORMAT (1H1)
100   RETURN
      END
```

Заметим, что единственное средство использовать свободные переменные в строгом соответствии со стандартом ФОРТРАНа – это размещать их в областях *COMMON* (IV.5.3). Это обычная техника, когда некоторое число свободных переменных обобществляется несколькими подпрограммами.

ФОРТРАН не разрешает пользоваться массивами, границы которых фиксируются уже при выполнении. В частности, важно не путать возможность, которую имеет программист для того, чтобы не фиксировать границы массива, являющегося формальным параметром подпрограммы (IV.4.5), с «динамическими» массивами, разрешенными другими типами распределения памяти. В некоторых случаях, однако, эта возможность может быть использована как заменитель: не имея полномочий параметрически задавать размер локального в подпрограмме массива, вызывающей программе поручают заботу о поставке в качестве параметра массива, границы которого она сама фиксирует.

IV.6.4. АЛГОЛ W

Метод, используемый АЛГОЛом W – динамическое распределение. Действительно, его можно применить не только к процедурам, но и к каждому блоку. Понятие области действия идентификаторов, которое рассматривалось в гл.III как характеристика структуры блоков, распространяется, таким образом, на *динамическую структуру* программы: не только объект (переменная или массив) определяется исключительно в блоке (или процедуре), где употреблено объявление, или во внутренних по отношению к нему блоках, но в той же мере в ходе выполнения каждый объект будет иметь различные «экземпляры» при каждом выполнении блока (или процедуры), которому он принадлежит. Таким образом, переменные в АЛГОЛе концептуально *«динамические»,* а не свободные.

В АЛГОЛе W, как видно, можно использовать массивы, границы которых фиксируются заново при каждом выполнении блока или процедуры, где они объявлены. Единственное условие состоит в том, чтобы значения всех переменных элементов, участвующих в вычислении границ, были известны в момент активации блока: это либо переменные, объявленные во включающем блоке и получившие

значения, либо параметры в случае процедуры. Вот типы программ, использующих это свойство:

(1)
```
BEGIN
INTEGER N;
READON (N);
WHILE N >= 0 DO
      BEGIN
      INTEGER ARRAY T(0 : : N);
      ...
      ...
      READON (N)
      END
END
```

(2)
```
BEGIN
...
      PROCEDURE A (INTEGER VALUE M, N);
      BEGIN
      INTEGER ARRAY S(–M :: N, 1 :: M + N);
      ...
      ...
      END;
A (27, 32);
END.
```

Заметьте, что при выполнении будет обнаружена ошибка, если $–M > N$ или $M + N < 1$. Неправомерен следующий пример:

(3)
```
BEGIN
INTEGER N;
INTEGER ARRAY T(0 : : N);
...
END.
```

Действительно, переменная N должна быть объявлена и инициализирована во внешнем блоке.

Систематическое использование динамического распределения исключает возможность применения свободных переменных в АЛГОЛе W. Чтобы получить результат, эквивалентный эффекту свободных переменных, надо объявить переменную в блоке, включающем как блок, предназначенный для повторных выполнений, так и блоки, содержащие его вызовы, если речь идет о теле процедуры. Пример:

```
BEGIN
    ...;
        BEGIN COMMENT: БЛОК, СОДЕРЖАЩИЙ ВСЕОБРАЩЕНИЯ К
                              ПРОЦЕДУРЕ "INCREMENTER";
        INTEGER COMPTEUR;
        PROCEDURE INCREMENTER;
                          COMPTEUR := COMPTEUR + 1;
        COMPTEUR : = 0; ...
        INCREMENTER; ... INCREMENTER; ...
        ...
                  BEGIN ... INCREMENTER; ...
                  END;
        END
...
```

END.

Отметим как анекдот, что в АЛГОЛе 60, который был предшественником АЛГОЛа W, была предусмотрена возможность использования массивов с «динамическими» свободными границами! Конечно, написание первых трансляторов наталкивалось на несовместимость и нужно было выбирать между динамическим распределением и свободными переменными.

IV.6.5. ПЛ/1

В ПЛ/1 программист имеет выбор для каждого объекта (переменной или «структуры», определяемой в следующей главе) между статическим распределением и динамическим распределением. Один из «атрибутов» (см. гл. II) каждого объекта является, в самом деле, атрибутом распределения, который может иметь значения:

– *STATIC* (статическое распределение)
– *AUTOMATIC* (динамическое распределение)
– *BASED*
– *CONTROLLED* (последние два атрибута будут рассмотрены в следующей главе)

• Всякий объект, объявленный *STATIC,* управляется статическим распределением: поэтому он будет свободным. И наоборот, всякий массив *STATIC* должен быть объявлен с постоянными границами.

• Всякий объект, объявленный *AUTOMATIC,* управляется динамическим распределением, и, следовательно, правила его использования в точности те же, что и в АЛГОЛе W. В частности, массивы могут быть объявлены с непостоянными границами, лишь бы все элементы, встречающиеся в определении этих границ, были объявлены и инициализированы во включающем блоке.

Определение атрибута распределения появляется в *DECLARE:*

```
DECLARE T CHARACTER (20) VARYING STATIC,
        R FLOAT (12) DECIMAL AUTOMATIC;
```

Все–таки значением «по умолчанию» этого атрибута является *AUTOMATIC,* так что

```
DECLARE R FLOAT (12) DECIMAL;
```

эквивалентно объявлению второй переменной в предыдущем примере.

Сосуществование «статических» переменных и «динамических» переменных может оказаться необходимым в программе. Ниже приведен пример (позволяющий показать интересный алгоритм), в котором программа ПЛ/1 печатает простые числа, не превосходящие ее аргумента *N,* ограничиваясь при этом теми, которые не были напечатаны при предыдущем вызове. Поэтому нужно предусмотреть локальную свободную переменную, например *NMAX,* инициализируемую нулем и задающую наибольшее ранее встретившееся *N.* Алгоритм, используемый для нахождения простых чисел, не превосходящих заданного *N,* известен с античных времен под названием «решето Эрастофена». Его идея состоит в установлении соответствия между каждым числом от 2 до *N* и двоичным состоянием «хороший» или «плохой» (отсюда динамическое распределение – необходим массив, границы которого зависят от *N).* Первоначально все числа «хорошие». На каждом этапе алгоритма берется наименьшее, еще не рассмотренное «хорошее» число; оно будет простым (это свойство, доказательство которого мы оставляем читателю, является «инвариантом цикла»). Все кратные ему числа «вычеркиваются» – отмечаются как «плохие», потому что они не могут быть простыми.

ПЛ/1

```
ERATO PROCEDURE (N); DECLARE N BINARY FIXED (IS, 0);
/*   ОТПЕЧАТАТЬ ВСЕ ПРОСТЫЕ ЧИСЛА, НЕ ПРЕВОСХОДЯЩИЕ N, КОТОРЫЕ
     ЕЩЕ НЕ БЫЛИ ОТПЕЧАТАНЫ */
/*   МЕТОД–РЕШЕТО ЭРАСТОФЕНА */
     DECLARE NMAX STATIC BINARY FIXED (15, 0) INITIAL(0);
/*НАИБОЛЬШЕЕ ИЗ РАНЕЕ ВСТРЕТИВШИХСЯ ЗНАЧЕНИЙ N */
          IF N >NMAX THEN
             BEGIN
             DECLARE BON(2 :N) BIT(1);
                DO 1 = 2 TO N :/* ИНИЦИАЛИЗАЦИЯ :ПОМЕ–
                       ТИТЬ ВСЕ ЧИСЛА КАК «ХОРОШИЕ» */
                       BON(I)= '1'B; /* «ИСТИНА В ПЛ/1
                       BON–«ХОРОШИЙ» */
                       END;
                PUT LIST ('ПРОСТЫЕ ЧИСЛА ОТ' NMAX + 1,
                'ДО', N);
                IF NMAX <= 2 THEN PUT LIST (1, 2);
                DO I = 3 STEP 2 UNTIL N;
                   IF BON (I) THEN
                         DO; /* I–ПРОСТОЕ: НАПЕЧАТАТЬ
                         ЕГО И ВЫЧЕРКНУТЬ ЕМУ
                         КРАТНЫЕ, ОТМЕЧАЯ ИХ КАК
                         «ПЛОХИЕ» */
                   IF I NMAX THEN PUT LIST (I);
                   DO J = I**2 TO N BY 2*I
                      /* КРАТНЫЕ, МЕНЬШИЕ I**2, БЫЛИ
                         УЖЕ ВЫЧЕРКНУТЫ И
                         СООТВЕТСТВУЮЩИЕ I ЧЕТНЫЕ
                         НЕ БУДУТ РАССМАТРИВАТЬСЯ*/
                         BON (J) = '0'B; /* «ЛОЖЬ» */
                         END;
                   END;
                NMAX = N;
                END;
          END ERATO;
```

IV.6.6. Реентерабельность. Многократное использование. Побочный эффект

Благодаря предыдущим обсуждениям мы теперь в состоянии дать определения трем ключевым словам, относящимся к подпрограммам:

Подпрограмма называется **реентерабельной,** если она способна оперировать только с областями памяти, принадлежащими вызывающим ее программам

Это условие исключает локальные переменные, передачи значением с переписыванием в область, принадлежащую подпрограмме, доступ к обобществленным переменным (общим областям и т.д.), но оно выполняется для подпрограмм, оперирующих, например, над «внешним стеком» (ср. ниже VI.3.4.3).

Смысл этого понятия в том, что реентерабельная подпрограмма может быть обобществлена между несколькими программными модулями, одновременно присутствующими в памяти (в мультипрограммной системе). Используется единственная копия подпрограммы.

Существует условие, близкое к реентерабельности, но менее сильное:

Подпрограмма называется **многократно используемой,** если состояние принадлежащей ей памяти одинаково до и после каждого из выполнений подпрограммы.

Такая подпрограмма оставляет память в том состоянии, в котором ее хочется увидеть, вернувшись. Смысл не в обобществлении, как при реентерабельности, а в возможности не «перезагружать» подпрограмму в память при многократных использованиях (отсюда ее название).

Побочный эффект подпрограммы состоит в модификации состояния вызывающей подпрограммы путем присваиваний элементам, отличающимся от явно передаваемых фактических параметров.

Подпрограмма, имеющая доступ к переменной в области *COMMON* и модифицирующая ее при чтении элемента из последовательного файла, производит побочный эффект. Побочный эффект может быть также результатом доступа к сложной структуре данных (гл. V).

Побочные эффекты усложняют понимание множества программ, и их следует избегать по мере возможности. Они особенно ощутимы в подпрограммах–«выражениях», делая недействительной аксиому Хоара для присваивания (III.4.1)

$\{P[E \rightarrow x]\}\ x \leftarrow E\{P\}$

в случае, когда выражение E приводит вызов подпрограммы к побочному эффекту.

IV.7. Расширения понятия подпрограммы

IV.7.1. Иерархическая структура вызовов подпрограмм

Представленное в этой главе понятие подпрограммы дает исключительно важное и мощное средство декомпозиции и «структурирования» программ. Тем не менее это понятие имеет свои границы.

Первая из этих границ есть строго иерархический характер вызовов подпрограмм. В классической концепции каждый момент выполнения отмечается активностью единственного программного модуля, «вызванного» другими через несколько уровней.

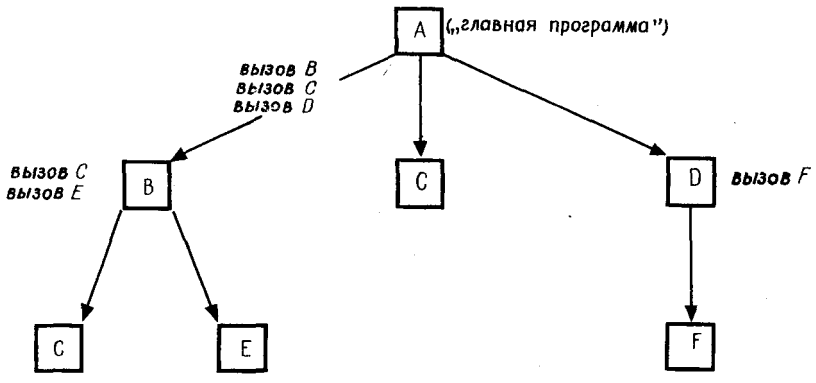


Рис. IV.10 Иерархическая структура вызовов.

Другими словами, состояние множества программ может быть охарактеризовано единственной точкой управления.

В некоторых задачах более результативным оказывается рассмотрение ситуации, где имеют место несколько автономных систем одного уровня, которые взаимно активируют друг друга, сами решают, «выключить» или «включить» одну по отношению к другой в том месте, где программа была перед этим прервана (а не систематически в начале). Такие системы называют скорее **сопрограммами**, нежели подпрограммами. Эти методы «самоуправляемого программирования» – применяются не только к задачам, где действительно необходимо изобразить или заказать несколько одновременно протекающих процессов (операционные системы, моделирование систем), но также в областях, которые могли бы показаться принадлежащими к последовательному или классическому программированию.

IV.7.2. Пример использования сопрограмм

Не очень глубоко развивая принципы «почти параллельного» программирования, которое выходит за рамки этой книги, мы тем не менее представим здесь введение в эту область с помощью одного примера (и второго, рассмотренного в упражнении IV.3). Используемый формализм основывается на понятиях языка СИМУЛА 67 [Дал 72a] [Биртуистл 73].

Программные модули «иерархического» способа, рассмотренные выше, общаются друг с другом с помощью двух операторов: вызов подпрограммы, обозначаемый ее именем; возврат подпрограммы к последнему вызвавшему ее программному модулю (неименуемому). Слова вызов и возврат были явно введены в ФОРТРАНе и ПЛ/1 (*CALL* и *RETURN)* и неявно в АЛГОЛе W и нашей алгоритмической нотации.

Этот способ общения остается применимым и к сопрограммам, но служит только для начальной активации совокупности сопрограмм и в заключение выполнения.

Между этой начальной активацией и завершающим возвратом сопрограммы взаимно активируются с помощью оператора, обозначаемого

возобновить c (x_1 x_2, ..., x_n)

где c – это имя сопрограммы, а x_1 x_2, ..., x_n – фактические параметры, которые ей передаются. Действие этого оператора, появляющегося в сопрограмме c, состоит в «выключении» c' и возобновлении выполнения сопрограммы c, начиная с оператора, непосредственно следующего за последним, выполненным ее оператором возобновить. Выполнение c' возобновится только тогда, когда некоторая сопрограмма выполняет оператор

возобновить c' (...)

Сопрограмма c в таком случае возобновит свое выполнение с первого оператора, следующего за ее последним выполненным оператором **возобновить**.

Напомним, что вызов подпрограммы возобновляет всегда выполнение с самого начала тела подпрограммы; в случае сопрограмм дело обстоит иначе, как это показано на Рис. IV.11 для случая двух сопрограмм, отвечающих друг другу.

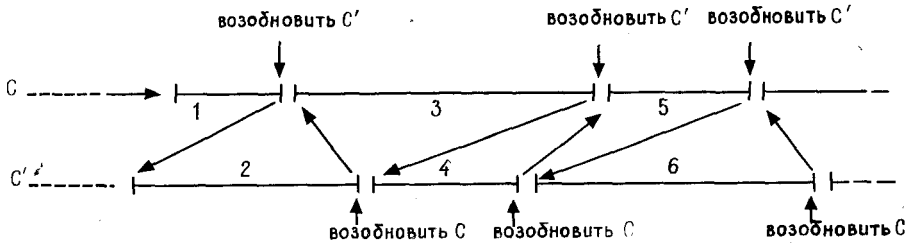


Рис. IV.11 Общение двух сопрограмм.

Другое важное отличие от схемы вызова подпрограмм – это симметрия «вызывающей программы» и «вызываемой программы», привносимая оператором **возобновить**. Не существует более иерархической структуры вызовов; есть множество сопрограмм одного уровня, которые могут активироваться одновременно. Если, однако, предположить существование «главной» программы, способной начать выполнение всей совокупности сопрограмм, то получается схема общения, представленная на Рис. IV.12, где возврат к главной программе может осуществляться любой из сопрограмм).

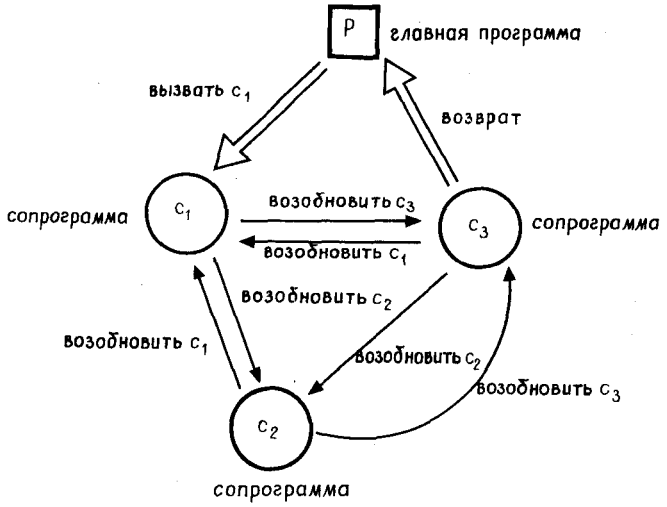


Рис. IV.12 Структура семейства сопрограмм.

Проиллюстрируем этот метод с помощью задачи, типичной для АСУ и трудно решаемой классическими методами (вложенными циклами); читатель может попытаться решить ее и без сопрограмм (рекурсивная формулировка тоже допускает элегантное решение этого типа задач). Речь идет об одновременном просмотре двух файлов; трудность вызвана, в частности, тем, что нужно учитывать случай, когда один из двух файлов уже исчерпан, а второй – еще нет (ср. также сложение двух разреженных массивов в цепном представлении, V.9.4).

Каждый из двух последовательных **файлов** F_1, и F_2 состоит из последовательности «записей»; каждая запись образована **ключом** c и связанным с ним **значением** v. Каждый файл «упорядочен по ключу», т.е. ключи последовательных записей расположены в возрастающем порядке с учетом некоторого отношения порядка; последовательные ключи могут оказаться равными; некоторые ключи могут встретиться в обоих файлах (Рис. IV.13). Требуется создать файл F_3, сформированный из записей, отсортированных в порядке возрастания ключей; каждая запись F_3 составляется из ключа c, встретившегося в F_1 или F_2, и значения v, равного сумме значений всех записей, имеющих с своим ключом в F_1 или F_2 (таким образом, все ключи записей в файле F_3 будут различными). Рис. IV.13 дает пример начальной ситуации

(файлы F_1 и F_2) и решения (файл F_3).

Используются три сопрограммы: сопрограмма, читающая файлы F_1 и F_2, сопрограмма «сравнения», выполняющая параллельное

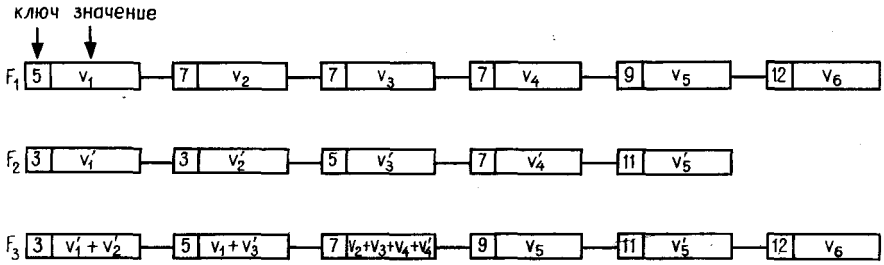


Рис. IV.13 Слияние с суммированием: пример и его решение.

сравнение в двух файлах, и сопрограмма «выхода», суммирующая итоги и записывающая их в файл F_3 (Рис. IV.14).

Предположим, что существует ключ, превосходящий все другие и не принадлежащий ни F_1 ни F_2. Его обозначение $+\infty$.

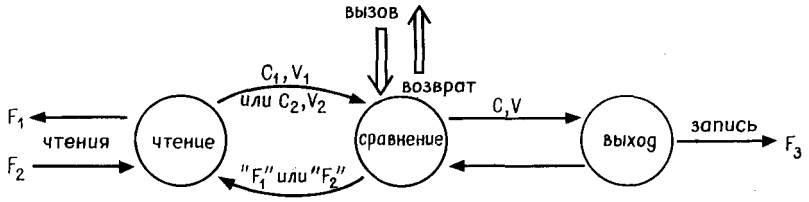


Рис. IV.14 Сопрограммы для слияния с суммированием.

Чтение записи [c, v] из файлов выполняется соответственно с помощью

читать (F_1) [c, v]

и **читать** (F_2) [c, v]

Такое чтение возможно, только если соответствующие условия

конец–файла (F_1)

и **конец–файла** (F_2)

отмечающие исчерпание файлов, имеют значение ложь. Занесение записей в файл осуществляется посредством

писать (F_3) [c, v]

Сопрограмма чтения имеет вид

```
сопрограмма чтение (аргумент f: ФАЙЛ;
                    результаты c : КЛЮЧ, v : ЗНАЧЕНИЕ)
  повторять бесконечно
    если конец–файла (f) то
      c ← +∞; v ← 0
    иначе
      читать (f) [c, v]
    возобновить сравнение
```

Сопрограмма **чтение** «читает» элемент в F_1 или F_2, если он есть, и передает управление сопрограмме **сравнение**. Заметьте – цикл **повторять бесконечно** характерен для этого метода программирования концептуально, **чтение** – это автономный бесконечный процесс; впрочем, практически предусматривается «останов».

Сопрограмма сравнение записывается в виде

```
сопрограмма сравнение
    переменные {свободные}  c1 c2: КЛЮЧИ,
                            v1, v2 ЗНАЧЕНИЯ;
    возобновить чтение (F1, c1 v1);
    возобновить чтение (F2, c2, v2);
    пока c1 ≠ +∞ или c2 ≠ +∞ повторять
        выбрать
            c1 ≤ c2 :  возобновить выход (c1, v1);
                       возобновить чтение (F1, c1, v1)
            c1 ≥ c2 :  возобновить выход (c2, v2);
                       возобновить чтение (F2, c2, v2)
    возобновить выход (+∞, 0)
```

Для большей симметрии относительно файлов (и тот и другой одинаково приоритетны, когда $c_1 = c_2$) было использовано понятие «недетерминированный выбор» **выбрать** (III.3.2.2). Конструкция

если $c_1 \le c_2$ **то** ... **иначе** ...

была бы тем не менее корректна здесь (она приводит к исчерпанию сначала всех ключей, равных c_1 в файле F_1, до поиска других равных ключей в F_2, когда обнаруживается $c_1 = c_2$; конечно, можно точно так же сделать противоположный выбор). Завершающий оператор **возобновить** выход(+∞, 0) нужен для «разблокирования» последней записи, предназначенной для занесения в F_3.

Сопрограмма выход суммирует значения данного ключа и выполняет занесение записи в F_3. Она должна сохранять в свободных переменных значение последнего отправленного ей ключа и суммировать соответствующие значения:

```
сопрограмма выход (аргументы c: КЛЮЧ, v: ЗНАЧЕНИЕ)
    переменные {свободные}  предыдущий–ключ: КЛЮЧ,
                            предыдущее–значение: ЗНАЧЕНИЕ;
    повторять бесконечно
        предыдущий–ключ ← c; предыдущее–значение ← v;
        возобновить сравнение;
        пока c = предыдущий–ключ повторять
            предыдущее–значение ← предыдущее–значение + v;
            возобновить сравнение;
        писать (F3) [предыдущий–ключ, предыдущее–значение]
```

Заметьте, здесь снова употреблен цикл **повторять бесконечно**.

Первоначально совокупность процессов запускается вызовом (типа подпрограммы) сопрограммы сравнение; завершается она **возвратом**, который следует за последним оператором в сопрограмме сравнение; в нашей нотации оператор возврата неявный. По поводу этого упражнения важно отметить характеристическое свойство программирования с помощью сопрограмм: чтобы хорошо понять каждую сопрограмму, важно рассматривать ее совершенно автономной, выполняемой «почти независимо», или «почти параллельно».

Идя несколько дальше, сопрограмму можно уподобить автономному процессу, похожему на «параллельные процессы», встречающиеся, например, в операционных системах. Операции типа возобновить получают тогда новый смысл: их можно рассматривать как *операции синхронизации,* с помощью которых процессы могут посылать сообщения другим процессам или, наоборот, ждать получения такого сообщения (не зная, впрочем, источника сообщения; такое неведение недопустимо при предыдущем формализме). Можно проверить, что операторы возобновить из вышеприведенного примера и упражнения IV.3 концептуально располагаются как раз между операциями послать сообщение и операциями ждать.

Надо добавить, что в общем случае передача информации не столь регулярна в том же смысле, как и для данной сопрограммы: операция возобновить то запрашивает результат, то поставляет данные – все это для одной и той же сопрограммы. Важно, следовательно, расширить для сопрограмм простые понятия параметров аргументы и результаты.

IV.7.3. Обсуждение и заключение

Сопрограммы дают решение проблемы децентрализованных «структур общения».

Другая проблема, плохо решаемая «классическими» подпрограммами, – это отношения между программными модулями и обрабатываемыми ими данными. ФОРТРАН, как мы видели, делает попытку в этой области, предлагая «модули данных» наряду с программными модулями; но отношение между *COMMON* и использующими его программами определено плохо. Еще один путь показывает язык СИМУЛА 67: класс СИМУЛЫ 67 (который представляет в этом языке также понятие сопрограммы) объединяет множество данных и подпрограмм, их обрабатывающих. В следующей главе будет показано, что такое группирование действительно соответствует описанию некоторого нового *типа* (см. также понятие *модуля,* которое будет развернуто в разд. VIII.3.4). Здесь же его можно рассматривать как объединение области *COMMON* и подпрограмм, имеющих доступ к ее элементам. Это приближение позволяет также решить некоторое число проблем, таких, как инициализация свободных переменных (осуществляемая программой, принадлежащей к тому же *классу,* что и переменные) и защита информации с помощью механизма, позволяющего доступ извне к информации, содержащейся в классе, только с помощью особых подпрограмм класса (подпрограмм–«информаторов»), предназначенных специально для этой цели: подобно всякой службе прессы, они показывают лишь то, что хотят показать. «От нас все скрывают, нам ничего не говорят» – возводится в принцип программирования!

УПРАЖНЕНИЯ

IV.1. Различные способы передачи

Указать значения a[1] и a[2] (возможно, неопределенные) после выполнения описанной ниже программы в случае, когда x передается

а) значением,

б) как результат,

в) как значение–результат,

г) по адресу,

д) по имени

```
переменная i: ЦЕЛ;
массив a [1 : 2] : ЦЕЛ;
i ← 1; a[1] ← 2; a[2] ← 3;
p(a[i])
```

с подпрограммой p, определяемой

```
программа p (.... x : ЦЕЛ)
    a[i] ← a[i] + 1;
    i ← i + 1;
    x ← x – 1
```

Предполагается, что целое i и массив a доступны подпрограмме p В АЛГОЛе W или ПЛ/1 это можно было бы реализовать, объявив p в главной программе (блочная структура), а в ФОРТРАНе – передав i и a в *COMMON*

IV.2. Повторение фактического параметра

В некоторых языках программирования запрещено вызывать подпрограмму с двумя или более параметрами разных типов. Можете ли вы оправдать это правило? Для всех ли способов передачи проблема ставится таким же образом?

IV.3. Чтение–компоновка–воспроизведение

(По [Дал 72a].) Дайте с помощью сопрограмм решение следующей задачи: прочитать с 80–колонных карт последовательность литер и переписать эти литеры в список с помощью устройства, печатающего 132 литеры в строке. При этом должны учитываться следующие правила:

- входные литеры обрабатываются как непрерывный поток, но с включением одного пробела после каждой карты;
- для всякой последовательности подряд расположенных пробелов печатается единственный пробел;
- символ "↑" заменяет всякое вхождение двух литер "**".

Чтение выполняется с помощью оператора

читать t

который читает одну карту и присваивает ее массиву (из 80 ЛИТЕР); этот оператор определен, только если условие конец–файла имеет значение **ложь**. Печать осуществляется посредством оператора

печатать *l*

где *l* – это массив из 132 ЛИТЕР

ГЛАВА V.
СТРУКТУРЫ ДАННЫХ

Как отыскать этот почтенный и таинственный, укрывающий шестиугольник? Был предложен старый метод: чтобы найти книгу A, надо обратиться сначала к книге B, которая указывает место A; чтобы разыскать книгу B, обращаются предварительно к книге C и так до бесконечности. Вот на такие–то приключения я сам расточал свои силы, потратил свои годы.

Хорхе Луис Боргес

Вавилонская башня, в
Вымыслах

СТРУКТУРЫ ДАННЫХ

В этой главе развивается систематическое применение анализа, намеченного в гл. III при обсуждении управляющих структур, ко второй фундаментальной компоненте программирования, «двойственной» по отношению к программам, – к данным. Говорить о структурах данных – уже значит утверждать необходимость методического и структурированного описания объектов, обрабатываемых программами. Результаты, достижение которых преследует методология, развернутая здесь и применяемая затем к пространному, но необходимому обзору некоторых из принципиальных, структур, составляют часть арсенала программиста.

Структуры данных – это *организованная информация,* которая может быть *описана, создана* и *обработана* программами. Именно благодаря структурам данных можно выразить содержательный смысл конкретных представлений информации на физических носителях тогда, когда сами по себе эти представления являются не более, чем аморфными образованиями битов.

Фундаментальные структуры данных были рассмотрены в гл.II; речь идет о базовых типах, существующих в языках программирования – ЦЕЛОЕ, СТРОКА, ЛИТЕРА, ВЕЩЕСТВЕННОЕ. Цель этой главы – дать методы построения и обработки более сложных структур данных, или *типов.*

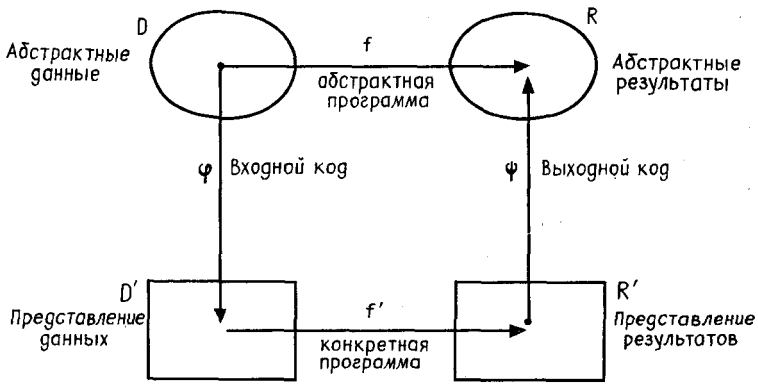


Рис. V.1 Обработка информации.

В общем контексте обработки информации (Рис. V.1) эта глава коснется не только проблемы соответствия между внешними данными и их машинными представлениями в передаче абстрактного алгоритма f конкретной программе f′ (т.е. проблемы построения входных и выходных кодов φ и Ψ, но и проблемы доступа программы к обрабатываемым ею данным.

Будем придерживаться в этой главе привычного соглашения, по которому слова *данные* и *информация* воспринимаются как синонимы. Напомним, что в строгом смысле «данные» – это только один из трех типов обрабатываемой программой «информации»; два другие – это *промежуточные переменные* и *результаты.*

V.1. Описание структур данных

V.1.1. Уровни описания

Рассмотрение структур данных часто затруднено весьма неудобным смешением абстрактных свойств структур данных и проблем представления, связанных с машиной или языком. Важно четко различать эти проблемы с помощью многоуровневых описаний.

Определение: Структурой **данных** называют набор из одного или нескольких *имен* и *множества* данных, к которым эти имена позволяют получить доступ.

В качестве примера рассмотрим информацию, представляющую состояние счетов вкладчиков в банке. **Физически** эти данные изображаются некоторым числом битов, байтов или слов, расположенных в оперативной памяти или на внешних носителях. **Логически** программисту важно обращаться к информации, связанной со счетом конкретного вкладчика, как к единому целому, образованному именем, например

СЧЕТ (МЕЙЕР)

над которым можно выполнять некоторое число точно определенных операций (приход, расход, вычисление остатка...).

Структура данных является, таким образом, одновременно и абстрактным объектом (одно или несколько имен), и физическим объектом (величины). Для того чтобы обеспечить соответствие между этими двумя крайними уровнями, программист должен задать декомпозицию структуры данных в терминах более элементарных структур. Всякая структура данных может описываться, таким образом, на трех различных уровнях, перечисляемых от более абстрактного к более конкретному:

- *Функциональная спецификация,* указывающая для некоторого класса имен **операции**, разрешенные с этими именами, и **свойства** этих операций; речь идет, следовательно, о **внешнем** определении.

- *Логическое описание,* которое задает декомпозицию объектов, отвечающую функциональному определению, на более элементарные объекты и декомпозицию соответствующих операций на более элементарные операции.
- *Физическое представление,* которое дает метод расположения в памяти вычислительной машины тех величин, которые составляют структуру, и отношений между ними, а также способ кодирования операций в языке программирования.

Три следующих раздела будут посвящены развитию этих трех понятий – функциональной спецификации, логического описания и физического представления. Заметим сразу же, что естественное желание при попытке сконструировать структуру данных для решения некоторой задачи состоит прежде всего в определении функциональной точки зрения (какие операции хотел бы я уметь выполнять над моими данными и каковы свойства этих операций?), затем логической точки зрения (какие данные известных типов и какие отношения, построенные для этих данных, позволят мне удовлетворить функциональные спецификации?), наконец, физической точки зрения (как мог бы я изображать и использовать все это в моей машине, в языке, которым я располагаю?).

Одной и той же функциональной декомпозиции могут, как мы увидим, соответствовать несколько логических описаний, которые в свою очередь могут реализовываться несколькими физическими представлениями. Следует, однако, быть уверенным в том, что декомпозиция каждого нового уровня достаточно хорошо «изображает» декомпозицию непосредственно превосходящего уровня; другими словами, анализ *функционального* уровня определяет спецификации, которые должны быть выбраны на *логическом* уровне и из которых точно таким же образом следует конструкция физического представления структуры данных.

V.1.2. Функциональная спецификация

Предположим, что нам как программистам поручено перестроение структуры данных, позволяющей банковским служащим, имеющим, например, доступ к диалоговым терминалам, выполнять некоторое число операций со счетами клиентов.

Первый этап состоит в точном определении этих операций; можно предположить, что служащему может, например, понадобиться выдать деньги со счета, узнать остаток счета, определить, является ли владелец счета кредитором, а также узнать фамилию вкладчика. Служащий более высокого ранга, скажем начальник отделения, мог бы, кроме того, открыть новый счет или принять деньги на существующий счет; это соответствует некоторой другой функциональной спецификации, и у нас будет еще повод к ней вернуться.

Итак, мы оказались перед задачей, состоящей в определении понятия СЧЕТ таким образом, чтобы впоследствии можно было именовать и создавать объекты типа СЧЕТ так же, как работают с объектами типа СТРОКА или ВЕЩ т.е. благодаря их абстрактным свойствам, а не сведениям об адресах их размещения в памяти или количестве битов, требуемых для их представления.

Определение структуры данных, таким образом, эквивалентно определению нового *типа* и его свойств.

> Такой же подход мог бы быть использован и для того, чтобы абстрактно определить некоторый известный тип. Так, тип «целое положительное или нуль» или ЦЕЛОЕ НАТУРАЛЬНОЕ определяется как множество элементов с некоторыми базовыми операциями: сложение, которое ставит в соответствие двум элементам этого множества их сумму x + y, вычитание; операция сравнения, обозначаемая ≤, которая двум элементам ставит в соответствие значение типа ЛОГИЧЕСКОЕ, истина или ложь. Тип ЦЕЛОЕ НАТУРАЛЬНОЕ может быть полностью определен заданием некоторого числа функций такого вида и некоторого числа свойств, таких,

как

$x + y = y + x$ (коммутативность)

$x \le y \Rightarrow (y - x) + x = y$ (сложение и вычитание являются обратными операциями по отношению друг к другу)

и т.д. Систематизация такого подхода в математике приводит к аксиоматическим определениям (аксиома Пеано для целых). Здесь мы внешне обобщим его на определение структур данных, или типов.

Вернемся к «банковскому счету»: тип СЧЕТ будет определен некоторым числом операций, которые объявляются с помощью нотации, сходной с той, что использовалась для подпрограмм:

- фамилия–вкладчика: СЧЕТ → СТРОКА

(операция, определенная над объектами типа СЧЕТ и дающая в качестве результата СТРОКУ – фамилию владельца счета);

- остаток: СЧЕТ → ЦЕЛОЕ

(операция, позволяющая узнать остаток счета. Предположим для упрощения, что суммы выражаются в сантимах и, следовательно, являются всегда целыми);

- кредитор: СЧЕТ → ЛОГ

(операция, позволяющая узнать, является ли вкладчик кредитором; смысл этого понятия подлежит уточнению);

- расход: СЧЕТ × ЦЕЛОЕ НАТУРАЛЬНОЕ → СЧЕТ

(операция, дающая новый счет, который получается из прежнего путем выдачи с него некоторой предполагающейся положительной суммы).

Для этих операций справедливы следующие свойства: для всякого счета c и всякого целого натурального x

(C_1) кредитор (c) ⇔ (остаток (c) ≥ 0)

(C_2) остаток (расход (c, x)) = остаток (c) – x
{операция «расход» выполняет именно то, что говорит ее имя}

(C_3) фамилия–вкладчика (расход (c, x)) = фамилия–вкладчика (c)
{операция «расход» не меняет фамилии вкладчика!}

Это последнее отношение показывает, что иной раз приходится при функциональной спецификации структуры данных уточнять свойства, которые могут не показаться необходимыми с первого взгляда.

Операции, встречающиеся в функциональной спецификации структуры данных, или типа, T, имеют вид

$$f : X_1 \times X_2 \times ... \times X_n \to Y_1 \times ... \times Y_m$$

где X_i и Y_i – типы; по крайней мере один из них должен быть T. Мы будем различать:

а) *функции доступа,* такие, что T появляется только слева от стрелки; они дают значения, характеризующие объекты типа T. Примеры: фамилия–вкладчика, остаток, кредитор;

б) *функции модификации,* такие, что T появляется слева и справа от стрелки. Они позволяют создавать новые объекты типа T, исходя из уже созданных объектов этого типа (и, возможно, других элементов). Пример: расход.

в) *функции создания,* такие, что T появляется только справа. Они позволяют создавать элементы типа T, исходя из элементов других типов (или из никакого элемента; слева от стрелки в этом случае ничего нет). Мы с ними еще не встречались.

Функциональная спецификация позволяет рассматривать структуру данных как абстрактный объект, не занимаясь проблемами конкретной реализации. Мы сможем в этой главе полностью характеризовать такие структуры, как «стеки», «файлы»,

массивы и т.д., не имея никаких гипотез ни об их декомпозиции на базовые объекты, ни тем более об их представлении в машине.

При доказательстве некоторых свойств иногда необходимо будет предполагать, что операции, встречающиеся в функциональной спецификации, являются единственно допустимыми над этими структурами. Пусть, например, тип T таков, что некоторое свойство P верно для всех результирующих объектов *функций создания,* встречающихся в спецификации T. Если любая *функция модификации* f, примененная к любому объекту t типа T, оставляет инвариантным свойство P, т.е. если

$P(t) \Rightarrow P(f(t, ...))$

то будем заключать, что P верно для всех объектов типа T. Этот метод доказательства оправдывается тем, что такие функциональные спецификации полностью описывают структуры.

В связи с *функциями модификации* может быть сделано одно возражение: функция расход, например, в том виде, как мы ее определили, не очень элегантна. Зачем включать в игру два счета, старый и новый? Ведь на самом деле хорошо известно, что результирующий счет операции «расход» тот же, что и начальный, но в нем просто изменился остаток; в терминах информатики расход имеет в качестве типа

расход: СЧЕТ × ЦЕЛОЕ НАТУРАЛЬНОЕ → ПУСТО

где первый аргумент есть модифицируемый параметр. Но при таком представлении становится трудным выразить свойства операций. На практике подпрограмма, представляющая расход, может не выдавать результата, а модифицировать свой первый параметр, рассматриваемый как модифицируемый параметр, если только отношения (C_1) и (C_3) проверяют значения всякого фактического параметра до и после вызова этой подпрограммы. В гл. IV было показано, что параметр типа модифицируемый параметр всегда разложим на аргумент и результат.

Заметим также, что можно было бы добавить условие

фамилия–вкладчика (c) = фамилия–вкладчика (c') ⇒ c = c'

(для всех счетов c и c'), чтобы распространить применение расход на операторы вида

c ← расход (c, x)

Вышеприведенная функциональная спецификация сознательно ограничена; она допускает только несколько строго определенных операций над существующими счетами: выдать с них деньги, узнать фамилию их владельца, определить остаток, узнать, являются ли вкладчики кредиторами или нет. В частности, не разрешено ни открытие новых счетов, ни вклады положительных сумм; эти операции можно было бы предоставить служащим более высоких иерархических уровней и сделать их, таким образом, объектом некоторой *другой,* более полной функциональной спецификации. В этой новой спецификации понятие «кредитор» могло бы быть более совершенным, можно учитывать положительный или отрицательный остаток клиента, предысторию его счета, состояние его кредитов и займов (взаймы дают только богатым) и т.д. Если это окажется возможным, попытаемся использовать одно и то же *логическое описание* (следующий раздел) для разных функциональных определений.

V.1.3. Логическое описание; смежность; разделение случаев; перечисление

Зафиксировав функциональную природу структуры данных, приходим к задаче представления ее в терминах базовых элементов и отношений между этими элементами.

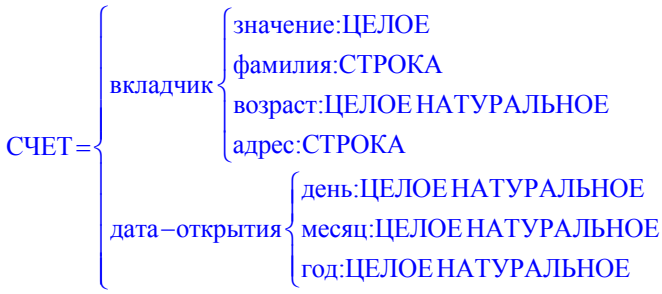
Определение структуры данных сводится, как мы видели, к созданию нового **типа**; до сих пор демонстрировалось, как такой тип описывается соответствующими ему операциями. Чтобы уметь создавать и обрабатывать объекты новых типов, нужно теперь дать определение этого типа в зависимости от ранее известных типов.

Эти «известные» типы включают типы, ранее определенные такими же методами, и некоторые *элементарные типы,* считающиеся априорно определенными. В примере банковского СЧЕТА «элементарными» типами могли бы быть

ЦЕЛОЕ
ЦЕЛОЕ НАТУРАЛЬНОЕ
ЛОГИЧЕСКОЕ {тип из двух элементов – **истина** и **ложь**}
СТРОКА {элементы этого типа представляют конечные последовательности литер}

Важно заметить, что выбор элементарных типов в известном смысле произволен. Тип, считающийся элементарным в некоторой задаче, в другой будет служить применением сложной структуры данных, разделяемой на элементы типов более низкого уровня. Так, для тех, кто занимается отладкой машины, ЦЕЛОЕ не является элементарным типом: это сложный тип, определяемый в зависимости от таких элементарных типов, как БИТ или БАЙТ; точно так же СТРОКА рассматривается в зависимости от обстановки либо как элементарный тип, либо как структура данных, получаемая из типа ЛИТЕРА.

Как описать СЧЕТ на логическом уровне? Можно использовать такую декомпозицию:



Это описание (по поводу которого интуитивно ясно, что оно задает *больше* информации, чем это необходимо для функциональной спецификации данных) указывает, что всякий СЧЕТ характеризуется одним ЦЕЛЫМ (суммой, связанной с этим счетом в текущий момент), тремя данными (одной СТРОКОЙ, одним ЦЕЛЫМ НАТУРАЛЬНЫМ и еще одной СТРОКОЙ), описывающими его владельца, и тремя данными, определяющими дату открытия счета. Каждое из сведений, входящих в описание СЧЕТ, имеет имя (например как значение, вкладчик, год и т.д.); если с –некоторый счет, то о различных составляющих его элементах, или «компонентах», можно говорить с помощью нотации, сходной с той, которая позволяет обозначать элементы массивов:

значение (с)
возраст (вкладчик (с))

Чтобы вышеприведенные описания можно было записывать в строку, воспользуемся скобками и разделителем «точка с запятой»:

```
тип СЧЕТ = (значение : ЦЕЛ;
            вкладчик : (фамилия : СТРОКА;
               возраст: ЦЕЛ НАТ; адрес : СТРОКА);
               дата–открытия: (день:ЦЕЛ НАТ;
                    месяц:ЦЕЛ НАТ; год:ЦЕЛ НАТ))
```

Как всякие системы обозначений, пытающиеся описать вложенность, две приведенные выше нотации (близкие к той, что предлагает ПЛ/1) достаточно тяжеловесны; за декларациями типов здесь трудно проследить. Поэтому предпочтительно, вообще говоря, ограничиваться единственным уровнем скобок, определяя промежуточные типы ЛИЧНОСТЬ и ДАТА :

тип ЛИЧНОСТЬ = (фамилия:СТРОКА; возраст:ЦЕЛ НАТ; адрес: СТРОКА)
{личность характеризуется двумя СТРОКАМИ и одним ЦЕЛ НАТ}
тип ДАТА = (день : ЦЕЛ НАТ; месяц : ЦЕЛ НАТ; год : ЦЕЛ НАТ)
{дата состоит из трех целых, не меньших нуля}
тип СЧЕТ = (значение:ЦЕЛ; вкладчик :ЛИЧНОСТЬ;
дата–открытия: ДАТА)

Если с – счет, то, как и раньше, можно использовать
значение (с) {которое играет роль переменной типа ЦЕЛ}
дата–открытия (с) {типа ДАТА}
день (дата–открытия (с)) {типа ЦЕЛ НАТ}

Итак, мы только что увидели первое средство определения типа в зависимости от других типов. Будем называть это средство **соединением,** обозначая его точкой с запятой: чтобы получить элемент типа СЧЕТ, нужно собрать ЛИЧНОСТЬ (точнее, ее «банковское» представление), ЦЕЛОЕ и ДАТУ (

Рис. V.2).

Рис. V.2 Счет.

Соединение – не единственная из операций над структурами данных. Уточним наше понятие СЧЕТА: «вкладчик» СЧЕТА может обрабатываться по–разному в зависимости от того, представляет ли он ЛИЧНОСТЬ или некоторую фирму. Тип ФИРМА можно определить, например, таким образом:

тип ФИРМА = (название–фирмы : СТРОКА; капитал: ЦЕЛ НАТ;
местонахождение : СТРОКА)

(капитал фирмы дает банкиру такую же уверенность в платежеспособности, как возраст личности). Таким образом, СЧЕТ определяется

тип СЧЕТ= (значение: ЦЕЛ; вкладчик : (ЛИЧНОСТЬ | ФИРМА);
дата–открытия: ДАТА)

где вертикальная черта (читаемая как «*или*») указывает разделение вариантов: вторая компонента СЧЕТА может иметь либо тип ЛИЧНОСТЬ, либо тип ФИРМА. Будем изображать эту операцию с помощью оператора **выбрать** ... (гл. III**)** и отношения **есть (является):**

выбрать
вкладчик (с) есть ЛИЧНОСТЬ : действие 1,
вкладчик (с) есть ФИРМА : действие 2 (1)

Действием 1 (и только им) обеспечивается доступ к полям фамилия (вкладчик)(с)), возраст (вкладчик (с)), адрес(с); Действием 2 (и только им) обеспечивается доступ к название–фирмы(вкладчик(с)) и т.д.

Будем предполагать, что всякий алгоритм, включающий обработку компоненты вкладчик счета, имеет структуру (1) или ей эквивалентную.

Остается рассмотреть третье правило композиции типов, которое, по правде говоря, является частным случаем «разделения вариантов»: речь идет о **перечислении**, позволяющем определить тип, который имеет конечное число возможных значений (как тип ЛОГИЧЕСКОЕ), задаваемых списком этих значений. Так, вместо того чтобы определять «месяц» в дате как ЦЕЛОЕ НАТУРАЛЬНОЕ, что обязывало бы каждый раз

проверять наличие значения «месяц», можно определить тип МЕСЯЦ перечислением, записав:

тип МЕСЯЦ = ("январь", "февраль", "март", "апрель", "май", "июнь", "июль", "август", "сентябрь", "октябрь", "ноябрь", "декабрь")

Запятые указывают перечисление. В более общем виде тип

определяется с помощью перечисления следующим образом:

тип T = (конст 1, конст 2, ..., конст n)

где конст 1, конст 2, ..., конст n представляют собой константы.

Требуется, чтобы все константы были одного и того же типа. Это ограничение нас не будет смущать. Оно могло бы показаться не изящным: в вышеприведенном примере мы обязаны оперировать со словом "январь" как со СТРОКОЙ, тогда как речь идет об *имени,* и буквы я, н, в и т.д. сами по себе нас не интересуют. Проблема состоит, однако, в том, что в таких языках, как ПАСКАЛЬ, где можно было бы писать (с несколько отличающимся синтаксисом):

тип *МЕСЯЦ = (январь декабрь)*

или **тип** *СВЕТ = (зеленый, красный, желтый)*

статут таких слов, как *январь* или *зеленый,* не очень ясен. Это не константы в привычном смысле, потому что целые, вещественные и другие константы имеют тип, который непосредственно выводится из способа их написания, тогда как в *зеленый* ничто не напоминает тип *СВЕТ.* Точно так же это и не идентификаторы, потому что нет никаких связанных с ними переменных. С другой стороны, никакая переменная не может быть объявлена с идентификаторами *зеленый* или *январь* после вышеприведенных объявлений.

Полное *объявление* типа СЧЕТ после всего сказанного записывается в виде

```
тип ЛИЧНОСТЬ = (фамилия: СТРОК А; возраст: ЦЕЛ НАТ; адрес: СТРОКА)
тип ФИРМА = (название–фирмы : СТРОКА; капитал: ЦЕЛ НАТ; местонахождение :
                СТРОКА)
тип ДЕНЬ–МЕСЯЦА = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,17, 18, 19, 20, 21,
                    22, 23, 24, 25, 26, 27, 28, 29, 30, 31)
тип МЕСЯЦ = ("январь", "февраль", "март", "апрель", "май", "июнь","июль", "август",
                "сентябрь", "октябрь", "ноябрь", "декабрь")
тип КЛИЕНТ = (ЛИЧНОСТЬ | ФИРМА)
тип ДАТА = (день : ДЕНЬ–МЕСЯЦА; месяц : МЕСЯЦ; год : ЦЕЛ НАТ)[1]

тип СЧЕТ = (значение : ЦЕЛ; вкладчик : КЛИЕНТ; дата–открытия : ДАТА)
```

Теперь уже можно оперировать объектами типа СЧЕТ:

переменные с, с', счет–дюпон: СЧЕТ

Заметим, что здесь выбран такой же прием, как в гл. III («Управляющие структуры»). Точно так же, как там указывался способ построения сложных программ из *элементарных действий* и *правил композиции* (цепочка, цикл, ветвление), здесь подобный принцип применяется к данным путем конструирования сложных типов из *элементарных типов* и правил композиции– соединения, разделения вариантов и перечисления.

Мы получили средство создавать новые типы и именовать *переменные* этих типов. Надо уметь также работать и с константами сложных типов. Для этого достаточно указать тип константы. Например, две константы:

```
ДАТА (19, "июль", 1980)
КЛИЕНТ (ЛИЧНОСТЬ ("ДЮПОН", 25, "БУЛЬВАР САН–МИШЕЛЬ, 13")
```

являются константами типов ДАТА и КЛИЕНТ соответственно.

Можно заметить также, что константа сложного типа немного похожа на вызов

[1] Не следует смущаться тем, что одно и то же имя (месяц) дано компоненте структуры и типу. Не касаясь этого как синтаксической проблемы, напомним, что мы предложили лишь простую нотацию, а не язык программирования.

подпрограммы: в случае типа, определенного соединением, «имя подпрограммы» заменено именем типа, а «фактические параметры» – значениями различных компонент, фигурирующих в константе (см. выше константу типа ДАТА); для типа, определенного разделением вариантов (как КЛИЕНТ), имеет место фактически «двойной вызов подпрограммы», уточняемый выбранным вариантом.

Понятие «константы» менее ясно для сложных типов, чем об этом заставляют предположить вышеприведенные примеры. Действительно, хотелось бы уметь писать:

переменные x:ЛИЧНОСТЬ, c:СЧЕТ, y:КЛИЕНТ;

x ← ЛИЧНОСТЬ ("ДЮПОН", 25, "БУЛЬВАР САН–МИШЕЛЬ, 13");

y ← КЛИЕНТ (x);

c ← СЧЕТ (30000, y, ДАТА (25, "март", 1981));

Вторая компонента выражения, присваиваемого c, связана с переменной x; поэтому доподлинно можно сказать лишь, что выражение является константой; чтобы указать структурированные «значения», которые могут принимать данные сложных типов, предпочтительнее говорить о *записи.* Заметим, что аналогия с подпрограммами может быть продолжена дальше: вторая компонента записи, присваиваемой c, могла бы быть вставлена «по значению», т.е. содержать *ссылку* на запись, указанную с помощью y. Но это делается на уровне физической реализации; мы еще вернемся к этому.

Логическое описание структуры СЧЕТ, очевидно, становится полным только в том случае, когда даны описания *операций* функционального определения в виде *подпрограмм,* работающих с объектами типа СЧЕТ и проверяющих постулированные в этом определении свойства.

Полное логическое описание структуры СЧЕТ показано на Рис. V.3. Заметим, что содержащиеся в нем подпрограммы позволяют делать больше, чем того требует функциональное определение разд. V.1.2; поэтому они могли бы быть использованы, чтобы отвечать более требовательным функциональным спецификациям.

Рис. V.4–другое возможное логическое описание, реализующее то же функциональное определение: оно предполагает, что в памяти хранятся 10 вновь введенных разных операций, вместо того чтобы корректировать «остаток» при каждой операции «расход» или «приход». Практически массив операции мог бы управляться скорее как *файл* (см. V.5); мы только хотели показать, что одному и тому же функциональному определению могут соответствовать очень разные логические реализации.

```
тип ЛИЧНОСТЬ          =   (фамилия : СТРОКА; возраст: ЦЕЛ НАТ; адрес:
                          СТРОКА);
тип ФИРМА             =   (название–фирмы : СТРОКА; капитал: ЦЕЛ НАТ;
                          местонахождение: СТРОКА);
тип ДЕНЬ–МЕСЯЦА       =   {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
                          20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31}
тип МЕСЯЦ             =   {"январь", "февраль", "март", "апрель", "май", "июнь",
                          "июль", "август", "сентябрь", "октябрь", "ноябрь",
                          "декабрь"}
тип КЛИЕНТ            =   (ЛИЧНОСТЬ(ФИРМА)
тип ДАТА              =   (день : ДЕНЬ–МЕСЯЦА; месяц: МЕСЯЦ; год : ЦЕЛ
                          НАТ);
тип СЧЕТ              =   (значение : ЦЕЛ; вкладчик : КЛИЕНТ; дата–открытия:
                          ДАТА);
программа открыть–счет: СЧЕТ (аргументы p: КЛИЕНТ, d: ДАТА,
                                                  v: ЦЕЛОЕ НАТУРАЛЬНОЕ)
    | открыть–счет ← СЧЕТ (v, p, d)
программа приход : СЧЕТ(аргументы c : СЧЕТ, x : ЦЕЛОЕ)
     {прибавить x положительное, отрицательное или нулевое}
    | приход ← c; значение (приход) ← значение (приход) + x
программа расход: СЧЕТ(аргументы c : СЧЕТ, x : ЦЕЛОЕ НАТУРАЛЬНОЕ)
    | расход ← приход (c, –x)
программа имя–вкладчика: ТЕКСТ (аргумент c:СЧЕТ)
    | выбрать
        | вкладчик (c) есть ЛИЧНОСТЬ:
                имя–вкладчика ← фамилия (вкладчик (c)),
        | вкладчик (c) есть ФИРМА;
                имя–вкладчика ← название–фирмы (вкладчик (c))
программа остаток: ЦЕЛОЕ (аргумент e:СЧЕТ)
    | остаток ← значение (c)
программа кредитор : ЛОГИЧЕСКОЕ (аргумент c:СЧЕТ)
    | кредитор ← (остаток (c)) ≥ 0)
```

Рис. V.3 Логическое описание СЧЕТА.

```
тип СЧЕТ = (число–операций : ЦЕЛ НАТ; массив операции [1 :10] .ЦЕЛ;
                имя:СТРОКА);
программа открыть–счет: СЧЕТ(аргументы n: СТРОКА, d : ДАТА, v: ЦЕЛОЕ)
    | массив t[1 : 10]: ЦЕЛ;
    | t[1] ← v;
    | открыть–счет ← СЧЕТ (1, t, n)
программа расход (модифицируемый параметр c:СЧЕТ; аргумент x : ЦЕЛ НАТ)
    | число–операций (c) ← число–операций (c) + 1;
    | если число–операций (c) = 11 то
    |       | для i от 2 до 10 повторять
    |       |       | операции (c) [1] ← операции (c) [1] + операции (c) [i]
    |       | число–операций (c) ← 2
    | операции (c) [число–операций (c)] ← –x
программа остаток : ЦЕЛ (аргумент c : СЧЕТ)
    | остаток ← 0;
    | для i от 1 до число–операций (c) повторять
    | остаток ← остаток + операции (c) [i]
{программы имя–вкладчика, кредитор, как на Рис. V.3}
```

Рис. V.4 Другое логическое описание СЧЕТА.

Рекурсивные определения

Исключительно полезная возможность в логических описаниях, не упоминавшаяся до сих пор, – это *рекурсивное определение:* в определении структуры T получаемой с помощью соединения, одна из компонент T может иметь своим типом то же самое T.

Модифицируем (еще раз) наш тип СЧЕТ, добавив ко всякому СЧЕТУ компоненту, называемую «гарантия», каторая представляет собой некоторый другой счет (предназначенный, например, для гарантии в случае катастрофы). Тогда записывают:

```
тип СЧЕТ = (значение : ЦЕЛ; вкладчик : КЛИЕНТ;
            дата–открытия : ДАТА; гарантия : СЧЕТ)
```

Если c есть СЧЕТ, то

- гарантия (c) это СЧЕТ
- дата–открытия (гарантия (c)) это ДАТА
- год (дата–открытия (гарантия (c))) это ЦЕЛОЕ НАТУРАЛЬНОЕ

и т.д.

Рекурсией называют наличие в определении объекта ссылки на сам объект. Соответствующее свойство называется *рекурсивностью.* Рекурсивность подпрограмм будет подробно рассмотрена в гл. VI.

Рекурсия может быть *косвенной:* тип T_1 определен как функция от типа T_2, который в свою очередь определен в зависимости от T_1; число уровней может быть произвольным.

> Косвенная рекурсивность, которая доставляет больше всего неприятностей авторам этой книги, это ... рекурсивность понятия рекурсии! Мы не могли уйти от включения в эту главу некоторого числа рекурсивных подпрограмм, которые, естественно вводятся в связи с рекурсивными структурами данных. Вообще говоря, ясно, что нельзя говорить о рекурсии (гл. VI), не показав предварительно структуры данных (настоящая глава); но, как можно достаточно серьезно раскрыть структуру данных до того, как проанализирована проблема рекурсии? В следующих разделах читатель сможет увидеть, насколько нам удалось реально применить рекурсивные структуры данных (указатели, взаимные ссылки...).

Определение

Логическое **описание** структуры данных, функциональная спецификация которых известна, включает:

а) Определение **типа** путем «логической идентичности», позволяющей включать ранее определенные типы и, возможно, сам определяемый тип (рекурсия), с использованием операций соединения, разделения вариантов и перечисления.

б) Некоторое число подпрограмм, оперирующих над определенным таким образом типом. Каждая операция функционального определения должна быть связана с одной из этих подпрограмм, проверяющих соответствующие свойства. Подпрограммы могут работатьс различными компонентами типа, определенного в **а)**; если компонента p определена *разделением вариантов,* т.е. если ее тип имеетвид $(T_1|T_2|...|T_n)$, то всякая подпрограмма доступа к компонентер объекта x должна содержать часть вида[1]

выбрать
| p(x) **есть** T_1: ... {обработка объектов типа T_1},
| p(x) **есть** T_2: ... {обработка объектов типа T_2},
| ...
| p(x) **есть** T_n: ... {обработка объектов типа T_n}

Чтобы закончить изложение логического описания структуры данных, упомянем наконец, что часто оказывается полезным определять структуру, включая в *разделение вариантов* специальный тип ПУСТО (*NIL* в языке ЛИСП, *NULL* в АЛГОЛе W или ПЛ/1 и т.д.), который позволяет «закончить» рекурсивную структуру. Например, если предположить, что СЧЕТ определен, как на Рис. V.4 (без рекурсивности), можно попытаться определить структуру данных, позволяющую работать с *несколькими* счетами. Тогда пишут:

тип СПИСОК–СЧЕТОВ = (ПУСТО|НЕ–ПУСТОЙ–СПИСОК–СЧЕТОВ)
тип НЕ–ПУСТОЙ–СПИСОК–СЧЕТОВ = (СЧЕТ; СПИСОК–СЧЕТОВ)

Это означает, что СПИСОК–СЧЕТОВ есть либо ПУСТО, либо СЧЕТ, соединенный с другим СПИСКОМ–СЧЕТОВ. Речь идет о рекурсивном определении; оно вновь возвращается к выражению того, что СПИСОК–СЧЕТОВ является соединением некоторого, возможно нулевого, числа СЧЕТОВ и пустого элемента.

Рекурсивные определения последнего типа, получаемые «разделением вариантов», где один из возможных вариантов–это ПУСТО, а другой–рекурсивный, дают, вообще говоря, более легко обрабатываемые структуры данных по сравнению с неограничиваемыми рекурсивными определениями, подобными приведенному на стр. 189, где не обеспечивается возможность избежать *циклических* структур, т.е.

гарантия$(c_1) = c_2$, гарантия$(c_2) = c_3$, ... гарантия $(c_{m-1}) = c_m$, гарантия$(c_m) = c_1$

Напротив, при определении, которое дано для СПИСКА–СЧЕТОВ, можно, используя некоторые гипотезы относительно выполняемых построений, обеспечить невозможность строить циклические списки. Хотя циклические структуры и оказываются иногда необходимыми, сведения о том, что структура не является циклической, вообще говоря, существенно упрощает изучение ее свойств.

V.1.4. Физическое представление. Понятие указателя

Предыдущий раздел показал, что при определении структур данных исходят из данных элементарных типов и отношений между ними. Будем полагать известными представления таких элементарных типов, как ЦЕЛОЕ, ЛИТЕРА и т.д.; поэтому будем интересоваться в этом разделе представлением отношений: соединение, разделение вариантов, перечисление.

[1] Когда не будет возникать двусмысленностей, мы будем предпочитать конструкции **выбрать** ... альтернативу **если ... то ... иначе ...** , если это будет упрощать запись.

Способы представления, описанные здесь, могут определяться либо программистом, либо транслятором. Смешения, возникающего в результате этого, избежать трудно: в зависимости от используемого языка программист может или не может рассчитывать на транслятор для решения некоторых задач представления. С этой точки зрения программист, использующий АЛГОЛ W или ПЛ/1, имеет бесспорное преимущество перед пользователем ФОРТРАНа. Однако и в том и в другом случае изучение методов представления полезно для более глубокого понимания алгоритмов.

V.1.4.1. Разделение вариантов, перечисление

Начнем с разделения вариантов и перечисления, которые, по существу, не составляют проблем. Пусть тип T определен разделением вариантов

$$\text{тип } T = (T_1|T_2|...|T_n)$$

где T_1 T_2, ..., T_n – известные типы. Объект типа T будет представлен двумя элементами:

- *индикатор типа:* целое i, $1 \le i \le n$, указывающее, какому T_i, принадлежит рассматриваемый объект;
- объект типа T_i.

Для представления индикатора типа достаточно $\lceil \log_2 n \rceil$ битов (где $\lceil x \rceil$ есть наименьшее целое, превосходящее или равное x)

Типы T_i, $i = 1, ..., n$, могут иметь представления, требующие память переменного размера. Эта проблема будет рассмотрена несколько ниже в связи с обсуждением рекурсивных структур данных. Самое элементарное решение состоит в систематическом использовании максимального размера.

- Объект типа T, определенный **перечислением:**

$$\textbf{тип } T = (c_1, c_2, ..., c_m)$$

будет попросту представлен индикатором – целым, заключенным между 1 и m, которое может разместиться в $\lceil \log_2 m \rceil$ битах.

V.1.4.2. Соединение

Соединение без прямой или косвенной рекурсивности не представляет больших трудностей: если T определено с помощью

$$\textbf{тип } T = (x_1 : T_,; x_2 : T_2; ...; x_n : T_n)$$

и если размер памяти, требуемой для представления объектов типов T_1 T_2, ..., T_n, известен, то размер памяти, необходимой для представления объекта типа T, определяется как сумма соответствующих размеров для объектов типов T_i. Так, на ИБМ 360/370 (ЦЕЛОЕ = 4 байта, ДВОЙНАЯ ТОЧНОСТЬ = 8 байтов, ЛИТЕРА = 1 байт) достаточно 13 байтов, чтобы представить объект типа **тип** ПРИМЕР = (e:ЦЕЛОЕ; d : ДВОЙНАЯ ТОЧНОСТЬ; c: ЛИТЕРА)

Допустим, напротив, что нам нужно работать со списками банковских счетов. Полагаем СЧЕТ определенным, как и раньше, без рекурсивности (следовательно, занимающим фиксированное место); определим типы

тип СПИСОК–СЧЕТОВ = (ПУСТО|НЕ–ПУСТОЙ–СПИСОК)
тип НЕ–ПУСТОЙ–СПИСОК = (СЧЕТ; СПИСОК–СЧЕТОВ)

СПИСОК–СЧЕТОВ – это множество счетов. Представление в машине таких множеств наталкивается на проблему их переменного размера: ничто не позволяет априорно зафиксировать предел. Можно, конечно, всегда использовать массив более или менее произвольного размера, что приводит к схеме

где первый элемент, целое n указывает число блоков, эффективно используемых в области, выделенной массиву. При таком решении, однако, нельзя гарантировать ни того, что эта область не окажется переполненной, если число СЧЕТОВ станет слишком большим, ни того, что не будет резервироваться бесполезное место в течение большей части времени.

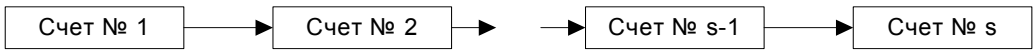
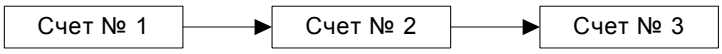


Рис. V.5 Список счетов.

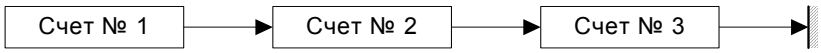
Вместо этого можно предложить список (Рис. V.5) как последовательность *нескольких* связанных между собой информационных блоков фиксированного размера; дело в том, что последовательность переменного числа блоков фиксированного размера часто бывает легче представить, чем фиксированное число блоков переменного размера. Чтобы суметь изобразить отношения между различными блоками, символизируемые стрелками на Рис. V.5, достаточно к каждому из них добавить данные о следующем элементе списка – его *адрес*.

> Такой адрес, используемый в представлении структуры данных, называется **указателем**. Его называют также *ссылкой.* Мы будем говорить, что ссылка (или указатель) *указывает* некоторое данное, чтобы выразить тот факт, что значение указателя есть адрес этого данного.

Мы видели, что специальный элемент, названный нами ПУСТО, играет особенно важную роль в рекурсивных структурах данных. В частности, он может служить для отметки конца списка таким образом, что список (или, более точно, «линейный список»; ср. V.6)



будет часто представляться как



где «заземление» изображает «указатель на ПУСТО». В машине такой указатель представляется специальным значением, которое не может быть адресом, например отрицательным значением.

> Так, предположив, что объекту типа СЧЕТ требуется 6 элементов памяти, а адресу нужен один элемент, можно использовать следующее представление (Рис. V.6):

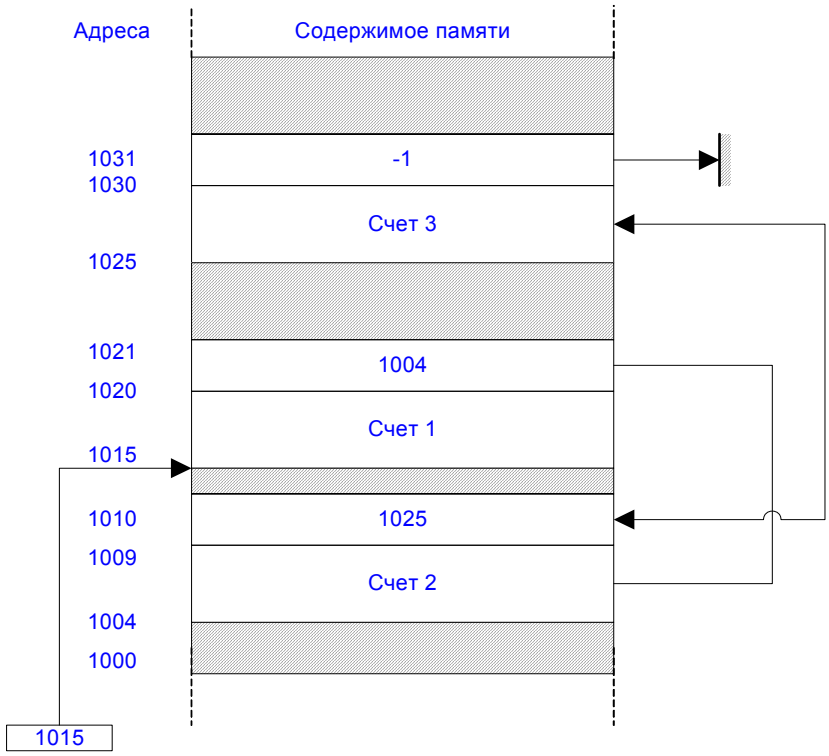


Рис. V.6 Представление линейного списка (счетов).

Список как единый именуемый объект становится в равной мере доступным благодаря указателю (значению 1015 в примере Рис. V.6). Однако, тогда как указатель, представляющий «внутреннее» отношение между элементами структуры данных (например, блоков списка), размещается непосредственно рядом с элементом, указатель, задающий точку входа в структуру, должен быть доступен программе и, значит, размещаться в месте, известном этой программе: указатель связан с некоторой переменной программы и требует в памяти места фиксированного размера, необходимого для представления адреса; этот размер известен при трансляции.

Таким образом, намечается решение проблемы управления объектами типов, требующих при выполнении памяти переменного размера, непредвидимого при трансляции; эта проблема касается одновременно рекурсивных структур данных и, как было показано в V.1.4.1, типов, получаемых «разделением вариантов»; она встречается в таких языках, как АЛГОЛ W, ПАСКАЛЬ, ПЛ/1, АЛГОЛ 68, предлагающих этот класс структур. Возможное решение такое: программа рассматривает объект сложного типа таким, который требует *фиксированного* размера памяти, необходимого для представления указателя и, возможно, индикатора (в случае разделения вариантов). Соответствующее пространство в памяти может выделяться классическим способом (статическим распределением, стеками и т.д.) как место, назначаемое переменной типа целого, вещественного и т.д.; рассматриваемые указатели будут обозначать адреса специальной области, называемой кучей и предназначенной исключительно для специальных структур данных непредвиденного размера (Рис. V.7).

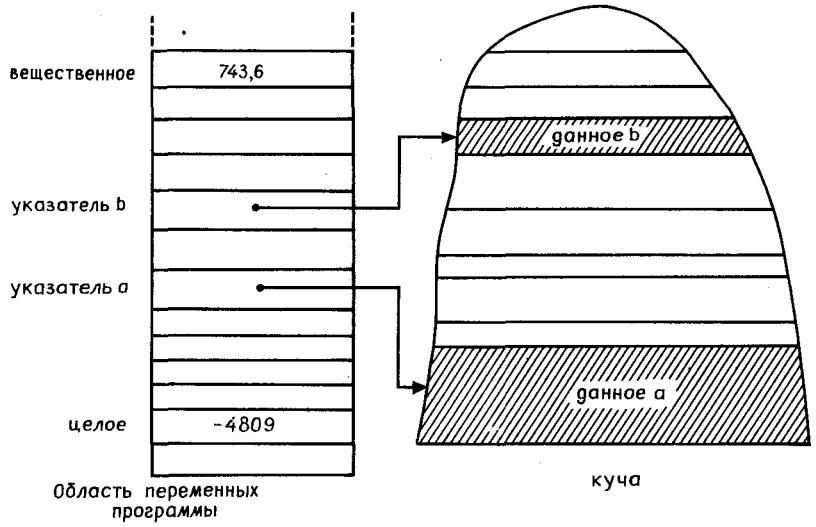


Рис. V.7 Использование кучи.

Управление кучей требует специальных методов. Основная проблема состоит в том, что некоторые элементы кучи становятся ненужными в определенные моменты времени, когда на них не направлен больше ни один указатель (указатели, которые были на них направлены раньше, изменили значение); можно попытаться освободить занимаемое ими место. Существуют два основных метода:

- связать с каждым элементом кучи некоторое целое, называемое **счетчиком ссылок** и равное числу указателей, направленных на него; счетчик ссылок корректируется при каждой операции; если его значение становится нулевым, то пространство, связанное с этим элементом, можно вновь использовать;
- разрешить заполнение кучи до тех пор, пока могут удовлетворяться запросы места; после этого вызывается специальная программа, называемая **сборщиком мусора,** которая исследует все доступные этой программе структуры данных, чтобы определить все занятые места и, следовательно, все свободные места в куче (более деликатные французы называют эту операцию *сбором крошек).*

Первый метод – метод счетчика ссылок – употребляется сравнительно редко, поскольку он требует достаточно тяжелого механизма счета. Второй метод ставит не меньше проблем; среди них одна из наименее сложных состоит в том, что программа сборки мусора должна размещаться в весьма ограниченной области памяти, так как она, по определению, вызывается, когда места уже почти не осталось!

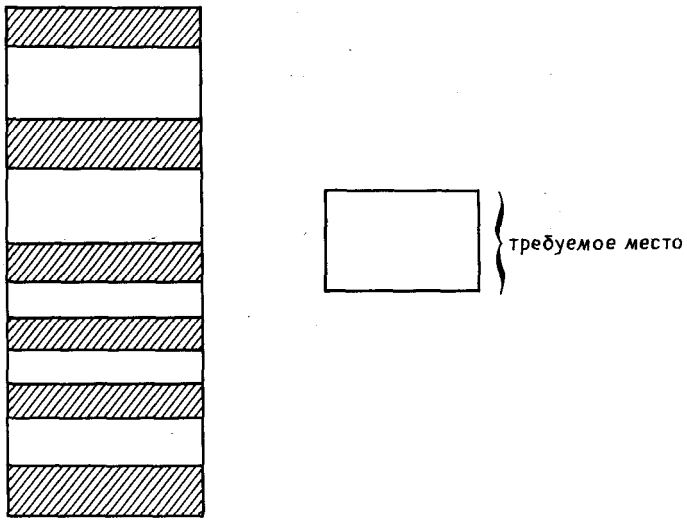


Рис. V.8 Дробление памяти.

Кроме того, феномен дробления памяти, в просторечии называемый «грюйеризацией»[1] и проявляющийся после нескольких вызовов сборщика мусора, делает невозможным вы деление «больших» блоков памяти, даже если суммарное свободное пространство достаточно для их размещения (Рис. V.8). Следовательно, нужно периодически «переупаковывать» память. По поводу всех этих проблем мы адресуем читателя к книге [Кнут 69]. Несколько дополнительных понятий, связанных со сборщиком мусора и их практическую реализацию, можно найти в разд. VI.3.5.2. Заметим, что метод кучи, который четко разделяет управление *переменными* программы и управление *структурами данных,* позволяет, вообще говоря, уйти от проблемы, называемой **висячей ссылкой:** «висячая ссылка» в языках с динамической блочной структурой, таких, как АЛГОЛ W или ПЛ/1 (IV.6.2), это ссылка на область памяти, которая фактически не подчиняется правилам динамического распределения. Эти проблемы будут еще раз затронуты в разд. VI.3.4.3, и обсуждение методов управления памятью будет дополнено.

V.1.5. Обсуждение. Проблемы

Мы умышленно настаивали на различиях трех уровней описания структур данных: функциональной спецификации, логического описания и физического представления. В действительности же эти уровни зачастую смешивают; однако нам кажется принципиальным при построении структуры данных отчетливо выделять:

- что нужно с ней делать;
- логическую (алгебраическую) структуру множества, позволяющего это сделать;
- машинное представление такого множества.

Логическая структура, позволяющая реализовать данное функциональное описание, редко бывает единственной, точно так же, как и соответствующее физическое представление. Выбор может осуществляться по нескольким критериям, в частности по критерию общности: делают так, чтобы одно и то же логическое описание могло обслуживать сразу несколько функциональных спецификаций; это можно было увидеть на примере «банковских счетов».

Мы представили построение структуры данных как некоторый *нисходящий* метод: исходим из концепций, направляясь к объектам, и рассматриваем, что *надо* сделать, прежде чем искать, *как* это сделать. Такой же метод, примененный к построению программ, был намечен в гл. III, и мы вернемся к нему еще в гл. VIII. Ясно, что на практике редко следуют чисто нисходящему методу: наш разум действует часто (хотим мы того или нет) методом проб и ошибок, продвигаясь вперед и отступая назад; кроме того, не всегда, «можно» сделать то, что «хочется», и задачи представления вынуждают иной раз вернуться назад, ломая спецификации. Надо добавить, что программист может оказаться в ситуации, где ему надо будет сдерживать себя в «восходящей» компоненте, когда он виртуозно пытается приспособить свою программу к существующим средствам, задуманным, быть может, для других задач, но достаточно общих, чтобы соответствовать более широким спецификациям: опыт каждого показывает, что спецификации, как и мир, изменчивы.

Языком программирования, открывающим путь современным концепциям структур данных, является, вне всяких сомнений, язык СИМУЛА 67, о котором уже шла речь в IV.7. В СИМУЛЕ 67 структура данных определена с помощью **класса;** класс, кроме всего прочего–мы видели, что он может играть роль «сопрограммы», – составляется: объявлением типа, похожим на наш уровень «логического описания» с указанием декомпозиции на элементарные поля; объявлением всех подпрограмм, способных работать с объектами этого типа, т.е. реализующих операции функциональ-

[1] В этом случае речь может идти, конечно, только о просторечии французов, которые часто употребляют «сыр грюйер» – сорт, отличающийся большим количеством дырок. – *Прим. перев.*

ной спецификации; наконец, последовательностью инициализаций, которые должны быть выполнены при создании любого объекта класса.

Если объявлен класс p с полями c_1, c_2, ..., c_n и соответствующими подпрограммами f_1, f_2, ..., f_n и существует объект o типа p, то можно создать новый объект и его значение присвоить o с помощью оператора вида

o := **новый** p(x_1, x_2, .., x_n)

где x_1, x_2, ..., x_n – значения, которые надо дать разным полям o. Этот оператор применяет к o стандартную инициализацию (соответствующую функции создания) и даже открывает возможность использования подпрограмм, обозначаемых $o.f_1$, $o.f_2$, ..., $o.f_n$ и воздействующих на o (и на их параметры, если они есть).

В СИМУЛЕ 67, однако, можно также иметь доступ к полям o, минуя соответствующие подпрограммы и отмечая $o.c_1$, $o.c_2$, ..., oc_n. В более поздних языках, базирующихся на СИМУЛЕ ([Лисков 76], [Вулф 75], [Лэмпсон 77]), это уже невозможно, и для того, чтобы иметь доступ к o, надо использовать функции внешней спецификации. Мы выбрали именно такой подход, потому что он дает существенные преимущества, отделяя представление алгоритмов от представления данных. Например, логические описания одной и той же структуры СЧЕТ, показанные на Рис. V.3 и Рис. V.4, полностью различны и представляют алгебраические структуры, никак не связанные между собой; но переход от одного описания к другому совершенно не влияет на программы управления счетами, использующие только *примитивы* фамилия–вкладчика, кредитор, остаток и расход.

Если, напротив, такие программы содержали бы непосредственно величины вида

фамилия (вкладчик (c))

или значение (c)

то эти программы следовало бы полностью переписать при необходимости перехода от одного описания к другому. Это обстоятельство само по себе оказывается зачастую достаточным для того, чтобы *не менять* описания, даже если за это говорят очевидные доводы, например эффективность.

Кроме того, понятие функциональной спецификации дает, как мы пытались это отметить, практическую основу для определения различных *прав доступа* по такой же логической структуре. Мы рассматривали пример прав доступа к СЧЕТУ, различающихся для обычного служащего и управляющего банком, но проблемы защиты структур данных ставятся шире, чем защита интересов вкладчика и банкира; это принципиально, в частности, в операционных системах ЭВМ.

Отметим, что столь точные функциональные определения, какие были введены, например, для СЧЕТОВ, быстро становятся весьма тяжелыми, как всякое аксиоматическое определение; для спецификации ФАЙЛА, например, уже требуются некоторые усилия. Поэтому вместе со строгими функциональными спецификациями или иногда вместо них мы часто будем использовать интуитивные, но более выразительные определения.

Наконец, не в пользу нашего метода следует добавить, что его нельзя применить с равным успехом ко всем структурам данных, которые бывает необходимо использовать. В частности, несколько ограничивающими оказываются *отношения,* которые могут встречаться в логическом описании: они не позволяют нам представить все структуры во всей их общности. Это говорит, однако, лишь о том, что наш формализм имеет свои границы, но его не следует отбрасывать: он позволяет и уяснить концепции, и улучшить программы.

V.2. Структуры данных и языки программирования

В этом разделе мы проведем четкое различие между АЛГОЛом W и ПЛ/1, с

одной стороны, которые предлагают программисту средства выражения, приближающиеся к уровню *логического описания* структур данных (V.1.3), и ФОРТРАНОМ, с другой стороны, в котором единственным средством структурирования данных является массив: проблемы представления, с которыми сталкиваются в этом языке, похожи на те, с которыми встречаются при программировании на ассамблере или машинном языке; в языках более высокого уровня такие проблемы находятся в ведении трансляторов.

Мы начнем с АЛГОЛа W, чтобы показать затем предлагаемые языком ПЛ/1 дополнительные варианты, и после сравнения этих двух языков перейдем к методам, используемым в ФОРТРАНе.

V.2.1. Записи и ссылки в АЛГОЛе W

В АЛГОЛе W существуют *данные типа* **«запись»** (*RECORD*), которые позволяют:

- непосредственно представить некоторые очень простые структуры данных;
- строить с использованием указателей более сложные структуры.

Запись в АЛГОЛе W – это способ *группирования элементарных данных* и, следовательно, представления главным образом *соединений.* Следует придерживаться двух правил:

а) число элементарных данных, встречающихся в записи, фиксировано ;

б) поскольку запись содержит только элементарные данные, она может иметь *только один уровень* структурирования.

Из условия а) вытекает, в частности, что запись не может ни содержать массивов (в АЛГОЛе W границы массива могут при некоторых условиях быть фиксированы при выполнении), ни представлять собою рекурсивную структуру данных. Первое из этих ограничений ставит иногда серьезные проблемы в АЛГОЛе W. Рекурсивные структуры, так же как и операция *разделения вариантов,* реализуются комбинированным использованием записей и ссылок, как это будет видно немного ниже.

Например, объявление записи *RECORD:*

```
RECORD PERSONNE (STRING NOM; INTEGER AGE;
                 STRING (20) ADRESSE)
COMMENT PERSONNE–ЛИЧНОСТЬ, NOM ФАМИЛИЯ,
AGE–ВОЗРАСТ, ADRESSE–АДРЕС;
```

говорит о том, что «родовое имя» *PERSONNE* (*ЛИЧНОСТЬ*) обозначает группирование текста из 16 литер[1] (названного *NOM – ФАМИЛИЯ*), целого (названного *AGE ВОЗРАСТ*) и 20–литерной строки (названной *ADRESSE – АДРЕС*). Запись *PERSONNE* представляет собой только модель группирования данных, некоторый новый «тип». Сами данные могут быть созданы по этой модели в любом числе экземпляров[2]. Их так и называют различными **экземплярами** записи. Один от другого экземпляры отличаются идентификаторами типа *REFERENCE* (ссылка), которые являются ни чем иным, как указателями.

Так, если желательно создать и раздельно использовать три экземпляра «структуры» *PERSONNE,* нужно объявить три идентификатора, например *QUIDAM_1, QUIDAM_2* и *QUIDAM_3,* и указать, что они могут обозначать группирования типа

```
REFERENCE (PERSONNE) QUIDAM_1, QUIDAM_2, QUIDAM_3,
COMMENT: QUIDAM – НЕКТО;
```

Видно, что в АЛГОЛе W физическое понятие указателя оказывает влияние на

[1] Напомним, что объявление *STRING S* эквивалентно *STRING (16) S*, т.е. длина «текстовой» переменной по умолчанию равна 16.

[2] Это число экземпляров ограничивается на практике, конечно, свободным местом в памяти, а не априорным объявлением.

логическое описание: объявленные *QUIDAM_1* и т.д. с логической точки зрения являются объектами типа *PERSONNE*, что соответствует физическому содержанию представляемых ими величин (адресов).

Во всяком блоке, в котором встречается это объявление, *QUIDAM_1*, *QUIDAM_2* и *QUIDAM_3* имеют целью указать три группирования данных, составленных по модели *PERSONNE*. Всякая попытка использовать их для указания данных, описанных другой *RECORD* (записью), выдаст сигнал об ошибке при трансляции.

Первоначально для этих трех ссылок только резервируется место, а сами ссылки ничего не означают.

(точно так же, как объявление *INTEGER X* создает не целое значение, а переменную, *способную принимать* целые значения).

Теперь можно задать значения *QUIDAM_1*, *QUIDAM_2* и *QUIDAM_3*, записывая, например,

QUIDAM_1 := PERSONNE ("ЮЛИЙ ЦЕЗАРЬ", 30, "КАПИТОЛИЙ")

что дает тройной эффект:

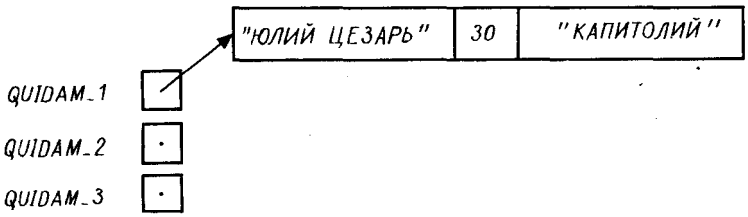
а) запросить у операционной системы машины «распределение» (т.е. получение и передачу в. распоряжение программы) области памяти, способной содержать группирование данных специфицированной модели, в нашем случае *PERSONNE;*

б) инициализировать содержимое этой области памяти указанными в скобках значениями, типы, которых должны соответствовать объявлениям записи (при нарушениях имеют место сообщения об ошибках трансляций[1]).

Это условие здесь удовлетворено: текст *"ЮЛИЙ ЦЕЗАРЬ"*, целое 30 и текст *"КАПИТОЛИЙ"* присваиваются в таком порядке трем частям *NOM*, *AGE*, *ADRESSE* нового экземпляра записи *PERSONNE*.

в) присвоить адрес начала этой области в качестве значения "ссылки" *QUIDAM_J*, указанной слева от знака ":= ".

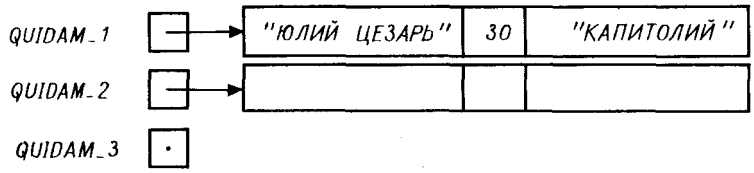
Выполнение этого оператора приводит, следовательно, к такой ситуации:



Не обязательно инициировать поля записи константами. Можно написать

QU1DAM_2 : = PERSONNE

после чего получают конфигурацию

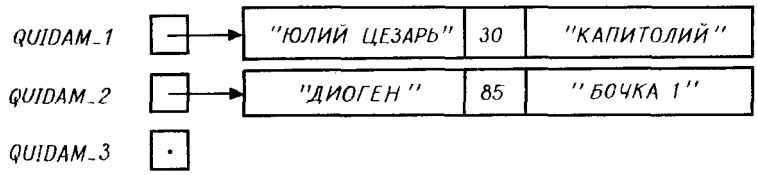


[1] Точнее, важно, чтобы содержащийся в выражении тип мог быть присвоен соответствующей компоненте по правилам гл. II. В частности, речь может идти о более коротком тексте; в дальнейшем мы будем говорить о *пробелах,* которые всегда могут быть дополнены справа.

Тогда можно индивидуально инициализировать различные компоненты *QUIDAM_2,* например

NOM (QUIDAM_2):.= "ДИОГЕН";
AGE (QUID AM_2); = 85;
ADRESSE (QUIDAM_2): = "БОЧКА 1"

что приводит к



В общем случае, если объявлена запись

RECORD ENREG (mun$_1$ COMPOSANT_1; mun$_2$ COMPOSANT_2; ....
mun$_n$ COMPOSANT_N)

и если вместе с тем объявлено

REFERENCE (ENREG) X

то элементы *COMPOSANT_1(X), COMPOSANT_2(X)* и т.д. играют такую же роль, как и переменные типов *mun$_1$, mun$_2$* и т.д. (подумайте об элементах массива).

Мы видели, что происходит, когда переменной типа «ссылка» присваивается выражение, включающее имя записи, например

QUIDAM_1 : = PERSONNE ("ЮЛИЙ ЦЕЗАРЬ", 30, "КАПИТОЛИЙ")

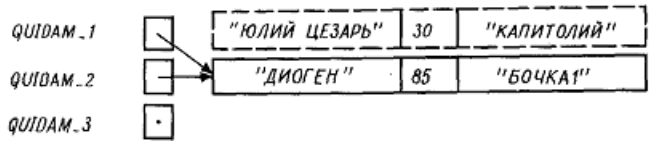
или *QUIDAM_2 : = PERSONNE*

В этом случае вычисление присваиваемого значения приводит к созданию *нового экземпляра* записи *PERSONNE*. Речь идет некоторым образом о «передаче значением» (ср. IV.4.2).

Можно присвоить также некоторой переменной, объявленной как *«REFERENCE* на тип записи», значение переменной того же типа, например

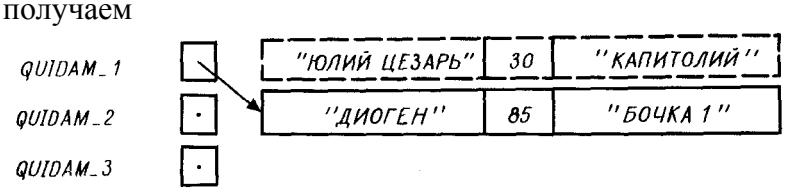
QUIDAM_1:= QUIDAM_2

В этом случае по аналогии с параметрами подпрограммы можно говорить о «передаче по адресу» (IV.4.4): никакая запись не переписывается; соответствующее *QUIDAM_1* значение указателя просто изменяется, с тем, чтобы указывать тот же экземпляр *PERSONNE,* что и *QUIDAM_2:*



Тогда запись ЮЛИЯ ЦЕЗАРЯ становится недоступной, «мертвой». Теперь может ставиться обсужденная в V.1.4 задача освобождения соответствующей области памяти с помощью *сборщика мусора.*

Существует совершенно особая «запись», значение которой можно присвоить любой переменной типа *REFERENCE(E)*, где *E* – тип произвольной записи: это константа *NULL* версия АЛГОЛа W константы ПУСТО. Выполняя *QUIDAM_2: = NULL* получаем



Присваивание константы *NULL* переменной *REFERENCE* может служить средством для того,

чтобы «убить» указываемый экземпляр записи–кроме ситуации, как в нашем случае, в которой после этого присваивания на него продолжает указывать некоторая другая ссылка.

Можно проверять, равны ли две *REFERENCE* или равна ли *REFERENCE* константе *NULL;* действительно, операторы отношений = и ¬= определены на объектах типа ссылка. Тогда в том месте, где мы остановились в нашем примере,

QUIDAM_1 = NULL есть **ложь** *(FALSE) QUIDAM_2 = NULL* равно *TRUE*

QUIDAM_3 = QUIDAM_2 равно *FALSE*

Важно, что если переменная типа *REFERENCE* становится *NULL,* то вопрос о доступе к компонентам указанной записи больше не ставится (указанной записи больше нет!). Так, всякая попытка определить

NOM (QUIDAM_2)

вызовет сигнал ошибки и останов программы.

Разделение вариантов реализуется в АЛГОЛе W не на уровне описания типов записей, а на уровне объявления переменных *REFERENCE.* Точнее, можно объявить

REFERENCE (ENREG1, ENREG2 ENREGN) X

где *ENREG1* и т.д. – это типы объявленных записей. Тогда *X* может указывать записи типа *ENREG1, ENREG2,* ..., или *ENREGN.* Это объясняет, почему надо уточнять имя записи, присваивая всякой переменной *REFERENCE* некоторый экземпляр записи

$X := ENREG1\ (x_1, x_2, \ldots x_m)$

Для того чтобы определить, какой тип записи указывается ссылкой в некоторый момент выполнения программы, можно пользоваться отношением *IS* (ср. **есть** в V.1.3). Например, имеют смысл логические выражения

REF IS ENREG1 REF IS ENREG2

Заметим, что

- у логического отношения *IS* нет отрицания; обратное к *X IS ENREG2* отношение можно, следовательно, записать только в виде ¬ *(X IS ENREG2)*
- когда *REFERENCE* имеет значение *NULL,* любое выражение вида *X IS RRR* имеет значение *FALSE,* какой бы ни была запись *RRR*

Напомним: для того чтобы определить, указывает ли ссылка на *NULL* достаточно логического выражения *X = NULL.*

Сложные типы, рекурсивность

Компоненты записи могут иметь своими типами любой элементарный тип. Тип *REFERENCE(ENREG),* где *ENREG* – тип записи, тоже считается элементарным: это значит, что с помощью записей можно представлять типы большой сложности. В качестве примера (ср. V.1.2 и V.1.3) можно определить:

```
RECORD PERSONNE (STRING NOM; INTEGER AGE; STRING (20)
        ADRESSE);
  COMMENT: ПЕРЕВОД ИМЕН В ПРИМЕРАХ СЛЕДУЮЩИХ СТРОК:
            RAISON_SOCIAL–НАЗВАНИЕ ФИРМЫ,
            CAPITAL–КАПИТАЛ,
            SIEGE_SOCIAL – МЕСТОНАХОЖДЕНИЕ,
            SOCIETE–ФИРМА,
            TITULAIRE–ВКЛАДЧИК,
            DATE_DE_CREATION – ДАТА_ОТКРЫТИЯ,
            COMPTE–СЧЕТ, VALEUR–ЗНАЧЕНИЕ,
            JOUR–ДЕНЬ, MOIS–МЕСЯЦ, AN–ГОД;
RECORD SOCIETE (STRING RAISON_SOCIAL; INTEGER CAPITAL;
                STRING (30) SIEGE_SOC1AL);
RECORD COMPTE (INTEGER VALEUR;
```

```
                    REFERENCE (PERSONNE, SOCIETE) TITULAIRE;
                    REFERENCE (DATE) DATE_DE_CREATION);
REFERENCE (COMPTE) C; CI, C2, C3, C4;
```

Тогда можно инициировать счета

```
C: = COMPTE (300, PERSONNE ("MEYER", 24, "AVENUE DU
                                        GENERAL"), DATE (29, 2, 86));
C1 : = COMPTE (3 000 000 000 0000, SOCIETE ("UNITED FRUIT",
                                10000000, "NEW YORK"), DATE (29, 2, 06));
```

или работать с их компонентами

```
VALEUR(C):=VALEUR(C1);
C2 : = C1; VALEUR (C2) : = VALEUR (C2) + 1 000;
TITULAIRE(C2) := SOCIETE ("PECHINEY", 10000, "PARIS");
```

и т.д.

Тот же механизм позволяет легко реализовать и рекурсивные структуры:

```
COMMENT : LISTE_DE_COMPTES – СПИСОК СЧЕТОВ,
                                    TETE – ГОЛОВА, SUITE – ХВОСТ;
RECORD LISTE_DE_COMPTES (REFERENCE (COMPTE) TETE;
                REFERENCE (LISTE_DE_COMPTES) SUITE);
REFERENCE (LISTE_DE_COMPTES) L, L1;
L := NULL; L : = LISTE_DE_COMPTES (C1, L)
```

Эти два оператора можно было бы заменить одним

```
L : = LISTE_DE_COMPTES (C1, NULL)
```

Напротив, присваивание

```
L: = LISTE_DE_COMPTES (C1, C2)
```

недопустимо: вторая компонента записи *LISTE_DE_COMPTES* должна иметь своим типом *REFERENCE (LISTE_DE_COMPTES)*. Теперь можно выполнить

```
L : = LISTE_DE_COMPTES (C, L);
L : = LISTE_DE_COMPTES (C2, L)
```

В таком случае L представляет список

Заметим, что в АЛГОЛе W элемент структуры данных (запись) не может сам

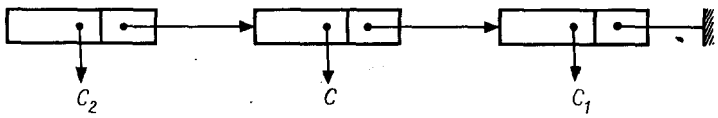


Рис. V.9

иметь сложную структуру (например, *СЧЕТ);* он может быть только ссылкой на такое данное (что выше выражено стрелками к C_2, C, C_1) или, разумеется, простым данным.

После выполнения вышеприведенного фрагмента программы *TETE(L)* имеет тип *COMPTE* и равно *C2* (физически его значение есть указатель, направленный на *C2*)

SUITE(L) имеет тип *LISTE_DE_COMPTES*

TETE (SUITE (L)) имеет тип *COMPTE* и равно *C*

TETE(SUITE(L)) = C логическое выражение, равное *TRUE*

L IS L1STE_DE_C0MPTES логическое выражение, равное *TRUE*

TITULAIRE(C) IS PERSONNE равно *TRUE*

TITULAIRE(C1) IS PERSONNE равно *FALSE*

TITULAIRE(C1)= TITULAIRE(C2) равно *FALSE*

SUITE(SUITE(SUITE)(L))) = NULL равно *TRUE*

SUITE (SUITE (SUITE (SUITE (L)))) не определено, и попытка вычисления вызовет останов программы.

Отметим, наконец, что тип *REFERENCE(E)* рассматривается как простой тип, т.е. существуют процедуры этого типа, массивы элементов этого типа, параметры процедур этого типа. В последнем случае, если специфицированы *VALUE, RESULT* и *VALUE RESULT,* то при каждом вызове будет переписываться только указатель, а не структура данных, им указываемая.

В разд. V.4 и следующих за ним будут даны многочисленные примеры использования данных типа *RECORD* и *REFERENCE.*

V.2.2. Структуры и указатели в ПЛ/1

Структура ПЛ/1 похожа на «запись» АЛГОЛа W. Существует, однако, важное отличие: объявление записи АЛГОЛа W никогда не определяет **объект,** а определяет только **«шаблон»** для класса объектов, которые могут создаваться в любом количестве в ходе выполнения программы; в ПЛ/1, напротив, «структура» предстает сама то как объект описываемого ею типа, то как «прототип», похожий на запись; и в том, и в другом случае на них могут быть обращены «указатели», похожие на *REFERENCE* АЛГОЛа W. Все зависит от *атрибута распределения памяти,* соответствующего объявлению структуры.

Прежде всего несколько синтаксических подробностей: структура сформирована из некоторого числа **«полей».** Каждое поле может быть определено как элемент «простого» типа (число, строка и т.д., как компонента записи АЛГОЛа W) и, кроме того, как *массив, а* также как *подструктура,* обладающая своими собственными полями: структуры ПЛ/1 могут, таким образом, включать несколько иерархических уровней.

В объявлении структуры каждому уровню присвоен номер: «корневой» уровень, обозначающий всю структуру в целом, имеет номер 1. Например, структура описания *PERSONNE,* соответствующая записи предыдущего раздела, может быть объявлена с помощью

```
DECLARE 1 PERSONNE,
          2 NOM CHARACTER (16),
          2 AGE BINARY FIXED,
          2 ADRESSE CHARACTER (20);
```

Каждому объявлению компоненты предшествует указание уровня. Если желательно уточнить понятия «возраста» (заменив его на «дату рождения») и «адреса», то необходимо включить отдельные объявления[1]*:*

```
/* VILLE–ГОРОД, RUE–УЛИЦА, NUMERO–НОМЕР_ДОМА */
DECLARE 1 PERSONNE,
          2 NOM CHARACTER(16),
          2 DATE,
             3 JOUR BINARY FIXED,
             3 MOIS BINARY FIXED,
             3 AN BINARY FIXED,
          2 ADRESSE,
             3 VILLE CHARACTER(10)
             3 RUE CHARACTER(20)
             3 NUMERO CHARACTER(3);
```

Что означает такое объявление структуры? Ответ на этот вопрос зависит от

[1] В ПЛ/1 для таких объектов, как «день», «месяц», «год», принимающих целые значения из определенной известной области, обычно вместо *BINARY FIXED* (целое) используют тип *PICTURE* (шаблон), которые мы не рассматривали.

атрибута распределения памяти, соответствующего структуре *PERSONNE*. Этот атрибут может иметь одно из четырех значений: «статическое» (*STATIC*), «автоматическое» (*AUTOMATIC*), «управляемое» (*CONTROLLED*) и «базированное» (*BASED*). Такой атрибут связывается только с «корневым» уровнем структуры, а не с элементами более высоких уровней. Если нет специальных указаний, как в нашем примере, то значение, принимаемое по умолчанию, есть *AUTOMATIC*, как для всех объектов ПЛ/1 (IV.6.5); в противном случае атрибут указывается. Например,

```
DECLARE I PERSONNE STATIC
    /* ИЛИ AUTOMATIC, ИЛИ CONTROLLED ИЛИ BASED(P)*/,
            2 NOM CHARACTER(16),
            2 DATE /* И Т.Д. */,
            2 ADRESSE /* И Т.Д. */,
```

Значения этого атрибута имеют такой смысл (мы мельком говорили о «статическом» и «динамическом» распределениях, которые рассматривались в гл. IV**):**

а) при **«статическом»** распределении объявление создает един ственный и фиксированный объект; это свободный объект в том смысле, что его–значение сохраняется от одного обращения к блоку или процедуре, которым он принадлежит, до следующего;

б) при **«автоматическом»** распределении точно так же определяется единственный объект, а не тип, но этот объект создается совершенно заново и должен, следовательно, вновь инициализироваться при каждом выполнении блока, в котором он объявлен. Заметим, что *STATIC* и *AUTOMATIC* эквивалентны для объектов самой внешней, главной процедуры, главной программы;

в) при **«управляемом»** (*CONTROLLED*) способе распределения объявление структуры не определяет никакого объекта, оно определяет тип; создание и ликвидация объектов этого типа полностью находятся в ведении программиста, которому приходится также заботиться о защите от возможных ошибок. Создание объекта осуществляется с помощью оператора *ALLOCATE* (выделить область памяти новому объекту), а ликвидация – с помощью оператора *FREE* (освободить область).

Фундаментальное свойство объектов, обладающих атрибутом *CONTROLLED*, состоит в том, что каждый оператор *ALLOCATE* создает его новую версию, которая *маскирует* предыдущую; последняя сохраняется, не имея права быть использованной. Оператор *FREE*, наоборот, уничтожает последнюю созданную версию и обеспечивает доступ к предыдущей, если она существует.

Пусть, например, структура *PERSONNE* объявлена с *CONTROLLED*. Создание объекта типа *PERSONNE* будет выполнено оператором

```
ALLOCATE PERSONNE;
```

Уничтожение последнего созданного объекта этого типа будет сделано путем

```
FREE PERSONNE;
```

После выполнения

```
ALLOCATE PERSONNE; ...; ALLOCATE PERSONNE; ...;
FREE PERSONNE;
ALLOCATE PERSONNE; ... ; ALLOCATE PERSONNE; ... ;
FREE PERSONNE;
```

будут созданы 4 «личности» и уничтожены две из них (вторая и четвертая из созданных); первая и третья еще останутся, но только третья будет доступна в программе. Именно ее обозначает имя *PERSONNE*. Оператор *FREE PERSONNE;* ее уничтожает; после этого единственно доступной остается первая из созданных.

Перед программистом встает, таким образом, задача: как запомнить «назначенные» и «освободившиеся» экземпляры, чтобы иметь доступ к нужному объекту путем некоторого числа операторов *FREE*? Можно использовать целый счетчик, инициализируемый нулем, увеличивающийся на единицу при каждом *ALLOCATE* и уменьшающийся на 1 при каждом *FREE*. ПЛ/1 предлагает также встроенную функцию *ALLOCATION* с одним параметром, имеющую результатом ЛОГИЧЕСКОЕ (*B1T(1)* в ПЛ/1): *ALLOCATION(X)* равняется *'1'B* (**истина**), если существует по крайней мере одна версия объекта *X,* и *'0'B* (**ложь**) в противном случае.

Например, чтобы уничтожить все существующие версии *PERSONNE,* можно было бы написать («всеобщая амнистия»)

```
DO WHILE ALLOCATION (PERSONNE);
        FREE PERSONNE;
        END;
```

и эта группа операторов будет давать желаемый эффект, даже если в начале не было создано никакой *PERSONNE;*

г) при «базированном» (*BASED*) способе, наконец, ситуация сходна с положением объектов типа *RECORD* в АЛГОЛе W. Как и в «управляемом» способе, объявление определяет только модель данных; объекты создаются фактически только по предписанию *ALLOCATE.*

Здесь, однако, данные доступны через соответствующий указатель; речь идет об объекте, объявленном с типом *POINTER*. Этот тип похож на тип *REFERENCE* (имя записи) в АЛГОЛе W с той важной разницей, что *POINTER* ПЛ/1 может служить для указания на объект любого типа; его не предназначают для объявлений какого–то одного определенного типа или множества типов.

«Указатель» объявляется с помощью

```
DECLARE P POINTER;
```

Указываемая структура объявляется атрибутом *BASED(P)* где *P* – указатель:

```
DECLARE     1 PERSONNE BASED (P),
            2 NOM CHARACTER (16)
            /* И Т.Д.*/;
```

Предписание *ALLOCATE PERSONNE* имеет в таком случае своим результатом создание «версии» структуры *PERSONNE,* на которую направлен указатель, связанный с ней с помощью объявления (Рис. V.10). Эта версия может быть поименована в программе с помощью обозначения

```
P –> PERSONNE
```

(где стрелка составлена из знака «минус» и следующего за ним знака «больше»).

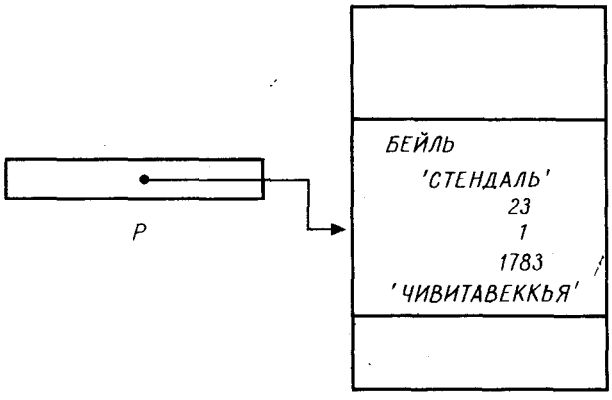


Рис. V.10

Можно оперировать присваиваниями объектов типа «указатель»; например, если

объявлено

```
DECLARE P1 POINTER
```

и если выполняется

```
P1 = P;
ALLOCATE PERSONNE;
```

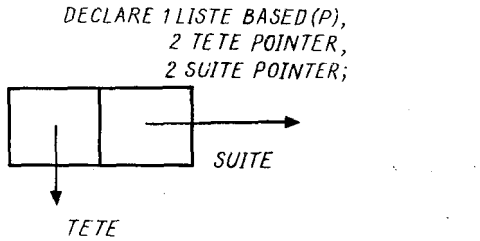
то создается новая личность, которую обозначает указатель *P:* действительно, будучи связанным объявлением с *PERSONNE,* он неявно участвует в присваивании при каждом новом *ALLOCATE PERSONNE.* Здесь предыдущая версия продолжает отмечаться указателем *P1*. Две доступные программе личности указаны соответственно *P1 –> PERSONNE* и *P –> PERSONNE.*

Если выполняется присваивание *P = P1*, то вторая личность больше не отмечается никаким указателем; она становится «мертвой» для программы и может стать жертвой сборщика мусора.

Тип *POINTER* – это совершенно полноправный тип; в частности, можно проверять равенство и неравенство указателей с помощью *P = Q* или *P ¬= Q*; равенство справедливо тогда и только тогда, когда *P* и *Q* указывают одну и ту же область независимо от содержания этой области. С другой стороны, никакой тип не связывается с указателем, и ПЛ/1 не предлагает эквивалента нашему отношению **есть** (АЛГОЛ W: *IS),* т.е. невозможно определить путем проверки тип данного, отмечаемого указателем. Если важно сохранить эту информацию, необходимо включить указатель *в* структуру, другой элемент которой, ЦЕЛЫЙ или ЛОГИЧЕСКИЙ, есть индикатор, задающий тип отмечаемого данного.

Как и в АЛГОЛе W, существует специальный пустой тип, здесь обозначаемый также *NULL;* например, при наших предположениях выражение *P –> PERSONNE = NULL* является ложью.

После того что сказано, должен стать ясным метод построения цепных структур данных; например, список может быть объявлен как структура



Здесь уточняется существенно меньше понятий, чем в АЛГОЛе W; *TETE* (голова) и *SUITE* (хвост) могут быть указателями, отмечающими все, что угодно, и программисту надлежит контролировать, используются ли они с достаточным основанием. Относительно «головы» можно было бы говорить определеннее, но и в этом случае нет возможности использовать существующий тип, например *PERSONNE,* если речь идет о списке личностей:

```
DECLARE 1 LISTE BASED(P)
        2 TETE /* PERSONNE */,
            3 NOM CHARACTER (16),
            3 AGE BINARY FIXED,
            3 ADRESSE CHARACTER (20),
        2 SUITE POINTER;
```

Здесь *SUITE* может указывать при выполнении все, что угодно, и никакой контроль при этом невозможен.

Доступ к компонентам структуры

Мы видели, как объявляют структуры, как создаются и уничтожаются соответствующие объекты. Нам остается описать средства доступа к компонентам

структур («полям»).

Нотация ПЛ/1 использует точку, располагая уточняемое имя перед уточняющим в противоположность АЛГОЛу W: там, где на АЛГОЛе W было бы написано

NOM (PERSONNE)

в ПЛ/1 обозначают

PERSONNE.NOM

Сравните: по–русски «адрес дома моего отца» и по–английски «my father's homme adress».

Тогда при последнем определении

```
DECLARE I PERSONNE/* ЗДЕСЬ АТРИБУТ РАСПРЕДЕЛЕНИЯ */,
          2 NOM CHARACTER (16),
          2 DATE
               3 JOUR CHARACTER (16), BINARY FIXED,
               3 MO IS BINARY FIXED,
               3 AN BINARY FIXED,
          2 ADRESSE,
               3 V1LLE CHARACTER (10),
               3 RUE CHARACTER (20),
               3 NUMERO CHARACTER (3);
```

PERSONNE.NOM имеет тип *CHARACTER(16)* и означает «имя», соответствующее «личности»;

PERSONNE.ADRESSE.NUMERO имеет тип *CHARACTER (3);*

PERSONNE.DATE – это подструктура структуры *PERSONNE,* сама обладающая тремя компонентами.

Такой способ обозначений называется уточняющими именами. Если *PERSONNE* имеет атрибут «статический» или «автоматический», такие имена означают компоненты единственного объекта *PERSONNE;* при «управляемом» способе – компоненты последнего экземпляра *PERSONNE,* если он существует; при «базированном» способе–компоненты экземпляра, отмечаемого указателем, который связан с *PERSONNE* с помощью объявления. В этом последнем случае можно получить доступ к компонентам *PERSONNE,* помечаемой другим указателем, с помощью такого обозначения:

```
P1 –>PERSONNE.DATE.MOIS /* MOIS–МЕСЯЦ */
```

Напомним, что компонента любого уровня структуры может быть массивом. Например,

```
/* ПЕРЕВОД ИМЕН В ПРИМЕРЕ:
     EMPLOYE            – СЛУЖАЩИЙ,
     NUMERO             – НОМЕР,
     SALAIRES           – ЗАРПЛАТА,
     SALAIRESJRUT       – ЧИСТАЯ ЗАРПЛАТА,
     SALAIRESSUPP       – ДОПОЛНИТ ЧАСЫ,
     PRIME              – ПРЕМИЯ,
     NOMBREENFANTS      – ЧИСЛО ДЕТЕЙ,
     AGEENFANTS         – ВОЗРАСТ ДЕТЕЙ*/
DECLARE 1 EMPLOYE /* ЗДЕСЬ АТРИБУТ РАСПРЕДЕЛЕНИЯ*/
            2 NUMERO BINARY FIXED,
            2 SALAIRES(12) /*ГОДОВАЯ ЗАРПЛАТА*/,
                 3 SALAIRE_BRUT BINARY FIXED,
                 3 HEURES_SUPP BINARY FIXED,
                 3 PRIME    BINARY FIXED,
```

```
2 NOMBREENFANTS BINARY FIXED,
2 AGE_ENFANTS (40) BINARY FIXED;
```

В этом случае уточняющие имена могут содержать указания индексов, например *EMPLOYE.AGE_ENFANTS(37)* (возраст 37–го ребенка), *EMPLOYE.SALAIRES(7).PRIME* (премия за июль). *EMPLOYE.AGE_ENFANTS* – это уточняющее имя, обозначающее массив из 40 элементов; *EMPLOYE.SALAIRES* – *это* массив из 12 подструктур.

В любом случае уточняющие имена играют полноправную роль переменных, массивов или подструктур некоторого типа. Можно поэтому к ним применить операции, определенные на этих объектах, и, в частности, присваивания.

Так, если при управляемом или базированном распределении создан новый объект с помощью

```
ALLOCATE PERSONNE;
```

то различным его компонентам можно дать значения присваиваниями

```
PERSONNE.NOM = 'БОРИС ГОДУНОВ'
PERSONNE.DATE.AN = 1598
```

и т.д. Напомним, что в АЛГОЛеW обычно связывают создание новой записи и использование ее компонент:

```
X : = PERSONNE («БОРИС ГОДУНОВ» и т.д.)
```

Заканчивая рассмотрение структур ПЛ/1, упомянем, что в случаях, когда исключается неоднозначность, уточняющие имена могут быть неполными: если существует только один объект типа *PERSONNE* (или если требуется иметь доступ к последнему созданному, или на объект направлен указатель, встречающийся в объявлении) и если никакая другая переменная не названа именем *JOUR,* то *JOUR* есть синоним *PERSONNE.DATE.JOUR* (или *P –>PERSONNE.DATE.JOUR).*

Со своей стороны, мы предпочитаем для большей ясности придерживаться полных уточняющих имен.

V.2.3. Сравнение АЛГОЛа W и ПЛ/1

Ясно, что ПЛ/1 предоставляет больше возможностей, чем

АЛГОЛ W:

- в АЛГОЛе W *REFERENCE* может быть направлена только на записи, тогда как в ПЛ/1 *POINTER* может обозначать любое данное. Кроме того, в АЛГОЛе W необходимо уточнять, какие типы записей могут быть избраны, чтобы служить «мишенью» конкретной *REFERENCE;* такое ограничение не имеет эквивалента в ПЛ/1;
- В АЛГОЛе W только модели данных типа *RECORD* имеют несколько «экземпляров» и могут выбираться «по заказу»; только таким образом их и можно использовать. В ПЛ/1, напротив, любой идентификатор может объявляться *BASED* и, наоборот, структура вполне может быть объявлена *AUTOMATIC.* Что касается атрибутов *STATIC* и *CONTROLLED,* у них нет эквивалентов в АЛГОЛе W. Отсутствие первого (свободные переменные) более ощутимо, чем отсутствие второго;
- использование в АЛГОЛе W пары, образованной *REFERENCE* и *RECORD,* в ПЛ/1 идентично работе с парой из *POINTER* и структуры *BASED.* Так, можно поставить в параллель следующие операторы:

АЛГОЛ W

```
RECORD A (INTEGER B;
          REAL C);
REFERENCE (A) Z;
.
.
.
Z := A;
.
.
.
.B(Z) := 1;
.
.
.
.Z := NULL;
.
.
.
```

ПЛ/1

```
DCL 1 A BASED (Z),
      2 B FIXED BIN,
      2 C FLOAT BIN;
.
.
.
ALLOCATE A;
.
.
.
Z –> B = 1;
.
.
.
Z = NULL;
.
.
.
```

что позволяет отметить очень большое сходство;

– в ПЛ/1 допускается многоуровневое представление и есть возможность выбора указателя по умолчанию; таких средств не существует в АЛГОЛе W.

Можно, таким образом, сказать, что эти возможности ПЛ/1 равны возможностям АЛГОЛа W, кроме того, у ПЛ/1 имеются и некоторые другие. И все же надо отдавать себе отчет в том, что возможности, имеющиеся в ПЛ/1 и отсутствующие в АЛГОЛе W, используются мало; их достаточно легко моделировать в конкретных условиях решаемой задачи.

Надо заметить, что ПЛ/1, давая программисту широкую свободу действий с указателями (и именно в силу этой свободы), приводит к серьезному риску ошибок. Это возникает вследствие возможного сосуществования данных, распределение которых управляется автоматически системой (данные *AUTOMATIC* и *STATIC)* и других данных, управление которыми лежит на ответственности программиста (данные *CONTROLLED* и *BASED).*

Это смешение делает программиста ПЛ/1 особенно уязвимым в проблеме *висячих ссылок,* т.е. всегда назначаемого указателя, который обозначает область, освобожденную системой или программистом. Немалому числу программистов эта проблема стоила бессонных ночей; менее претенциозные, чем ПЛ/1, языки, такие, как АЛГОЛ W, позволяют избежать ее. Об этой задаче управления памятью см. также разд. VI.3.4.3 и упражнение VI.3.

Заметим, наконец, что разница в обозначениях

JULES. AGE (ПЛ/1) и

AGE (JULES) (АЛГОЛ W)

хотя и не фундаментальна, все же существенна при «функциональ–нрм» подходе, в котором пытаются не решать слишком рано задачи представления. Действительно, первое обозначение с достаточно вескими основаниями предполагает представление личности *JULES* с помощью структуры; второе же, напротив, может с равным успехом указывать как поле записи, соответствующей *JULES,* так и применение подпрограммы (*PROCEDURE)* к *JULES;* важно, что эта двойная возможность сохраняется, и вопрос об эффективном представлении решается только на последующем этапе (ср. также разд. VII.1.2.4, «компромисс место–время»). С этой точки зрения можно сожалеть о

сделанном Виртом уже после АЛГОЛа W выборе в концепциях ПАСКАЛЯ, который возвращается к «базированной» нотации ПЛ/1.

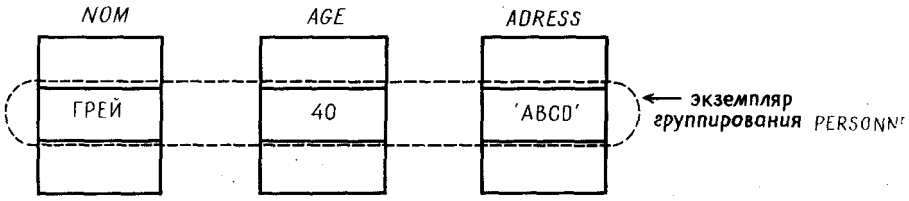
V.2.4. Методы ФОРТРАНа

Вопреки тому, что позволяет предположить заголовок «Методы ФОРТРАНа», все нижеследующее может прекрасно использоваться в АЛГОЛеW и в ПЛ/1: кто может многое, сумеет и малое! Но, с другой стороны, для ФОРТРАНа – это единственные возможности.

В ФОРТРАНе есть только один способ объявить несколько данных под одним именем: он состоит в том, чтобы построить из них массив. Если, кроме того, информация составлена из группы нескольких элементарных данных различных типов, то эту группу нельзя рассматривать в качестве единого объекта: надо строить столько массивов, сколько имеется типов элементарных данных. Так, для группирования данных *PERSONNE,* определенного в V.1.3 и составленного из трех данных *NOM, AGE* и *ADRESS*[1] должно быть объявлено

INTEGER NOM (100), AGE (100), ADRESS (100)

если не потребуется больше чем 100 экземпляров группирования *PERSONNE*. Каждый из этих экземпляров состоит из одного *NOM,* одного *AGE* и одного *ADRESS, имеющих один и тот же индекс.* В таком случае говорят о **параллельных массивах:** массивы имеют одинаковую размерность, и данные, представляющие часть одного группирования, находятся в одних и тех же позициях:



Это «распределение» данных по нескольким массивам является, несомненно, источником ошибок, так как программист обязан следить за использованием одинаковых индексов при доступе к разным элементам группирования, тогда как никакие «ошибки параллелизма» подобного рода не могут встретиться в методах программирования, рассмотренных в связи с АЛГОЛом W и ПЛ/1.

Другим неудобством является требование определить заранее максимальное возможное число экземпляров группирования данных для фиксирования размеров используемых массивов. Это ограничение может показаться очень обременительным, особенно когда приходится иметь дело с многочисленными структурами данных. Если попытаться резервировать для каждой из них максимальный размер, который она может принять при выполнении, то такой размер надо оценить. При этом возможен двойной риск: либо быть вынужденным резервировать слишком большую, но неиспользуемую область, либо подвергать вероятной неудаче программу из–за того, что единственная из используемых ею структур «переполнена», тогда как другие еще далеки от заполнения. Фактически же случается, что не все структуры достигают в одно и то же время своего максимального размера (Рис. V.11).

[1] ФОРТРАН не располагает переменными типа СТРОКА (гл. II). Мы предполагаем здесь, что фамилии или адрес изображаются кодом, представляющим собой целое. В примерах эти значения образуются четырьмя литерами.

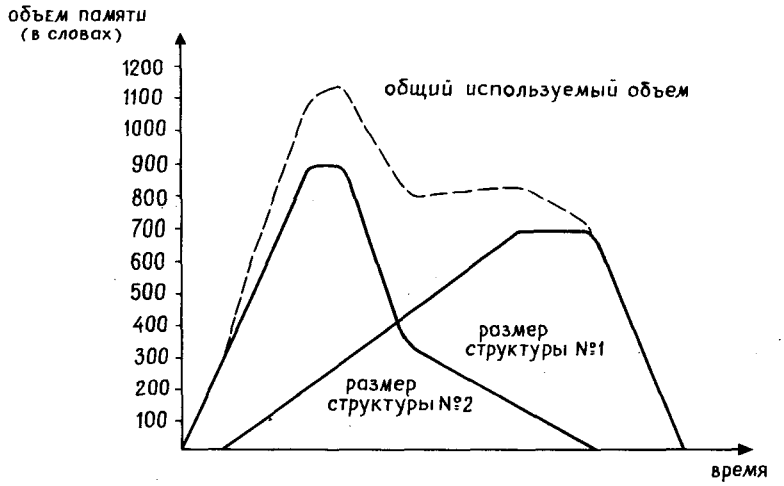


Рис. V.11

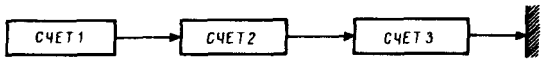
В таких ситуациях, когда используется много структур данных с меняющимся и трудно предсказуемым размером (например, в трансляторах), память управляется *кучей:* объявляется единый большой массив, куда помещают все объекты, размер которых не известен при трансляции.

В IV.5.3, где мы уже касались этой задачи управления памятью, была упомянута возможность использования для этой цели «общей области пробелов», которая в некоторых реализациях ФОРТРАНа представляет всю область свободной памяти; но это свойство реализовано не во всех системах.

Наконец, **в ФОРТРАНе не существует указателей.** Единственное средство представить значение, указывающее расположение принадлежащего некоторому группированию данного, состоит в использовании **индекса** этого данного в массиве, которому оно принадлежит. Это сводится скорее к относительному адресу (по отношению к началу массива), нежели к «абсолютному» адресу в памяти. Такой индекс представляется просто целым.

Таким образом, для представления в ФОРТРАНе структуры данных, включающей рекурсивное *соединение,* необходимо применить метод, подобный рассмотренному в разд. V.1 (V.1.4.2); просто здесь речь будет идти не об адресах, а об индексах массива (которые являются относительными адресами).

Рис. V.12 представляет структуру



такую же, как на Рис. V.6; чтобы представить указатель на ПУСТО, используется значение 0, потому что допустимые в массивах ФОРТРАНа значения индексов всегда больше или равны 1.

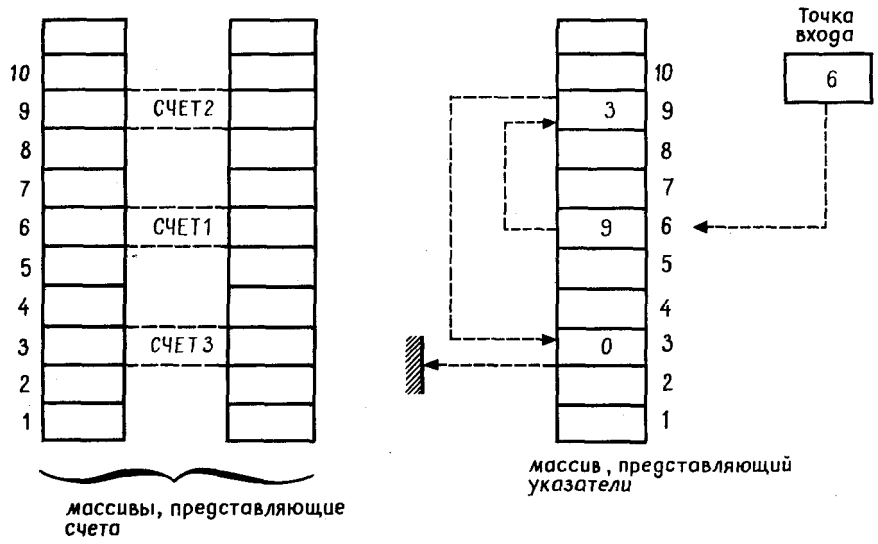


Рис. V.12 Соединение в ФОРТРАНе.

V.3. Множества. Введение в систематику структур данных

Мы приступаем ниже к индивидуальному описанию классических структур данных. Если эта часть главы названа «Введение в систематику[1] структур данных», то это не только потому, что в ней возникает много связанных с деревьями вопросов; это в первую очередь потому, что их изучение показывает строго определенную и совершенно линнеевскую классификацию способов размещения информации и доступа к ней.

Прежде чем погрузиться в свойства стеков, файлов, списков, двоичных деревьев, деревьев, графов и т.д., могут оказаться полезными некоторые размышления о более общей структуре данных, по отношению к которой все конкретные структуры данных выглядят ее ограничениями. Речь идет о структуре **множества** (иногда говорят – *таблицы);* множество – это просто группа данных, над которыми выполняется некоторое число операций, образующих функциональную спецификацию этой структуры.

Функциональная спецификация

Рассмотрим некоторый тип T. Определим тип, элементами которого являются множества объектов типа T, и обозначим его МНОЖЕСТВО$_T$. Можно представить себе необходимость выполнения следующих операций:

[1] В оригинале – «Введение в ботанику».– *Прим. перев.*

{создание}	создать–множество:	→	$МНОЖЕСТВО_T$ {операция без параметров, создающая пустое множество}
{модификация}	включить	$T \times МНОЖЕСТВО_T$	→ $МНОЖЕСТВО_T$ {сформировать новое множество, к которому добавлен один элемент}
{доступ}	найти:	$T \times МНОЖЕСТВО_T$	→ ЛОГИЧЕСКОЕ {проверить, принадлежит ли элемент T данному множеству}
{модификация}	убрать:	$T \times МНОЖЕСТВО_T$	→ $МНОЖЕСТВО_T$ {сформировать новое множество, из которого удален данный элемент. Операция имеет нулевой эффект, если элемент не принадлежал множеству}
{доступ}	пусто:	$МНОЖЕСТВО_T$	→ ЛОГИЧЕСКОЕ {проверить, есть ли элементы в множестве}

На этих операциях должны выполняться следующие свойства (для всех t, t' типа T и всех e, e' типа $МНОЖЕСТВО_T$):

(E1) пусто (создать–множество) {«создать–множество» создает пустое множество}
(E2) ¬ пусто (включить (t, e))
(E3) найти (t, включить (t, e)) {включаемый в e элемент есть в e}
(E4) ¬ найти (t, убрать (t, e)) {исключаемый из e элемент отсутствует в e}
(E5) $t \neq t' \Rightarrow$ найти (t, включить (f, e)) = найти (t, e) {можно переставлять включения разных элементов}
(E6) $t \neq t' \Rightarrow$ найти (t, убрать (t', e)) = найти (t, e)

Кроме этих действительно фундаментальных операций, в некоторых задачах могут встречаться и другие:

{доступ}	выборка: $МНОЖЕСТВО_T$	→ T {выдается элемент множества, если он существует, без удаления его из множества. Если множество пусто, то результат – ПУСТО}
{модификация}	выборка–удаление: $МНОЖЕСТВО_T$	→ $T \times МНОЖЕСТВО_T$ {выдается, кроме того, новое множество без удаленного элемента}
{модификация}	объединение: $МНОЖЕСТВО_T \times МНОЖЕСТВО_T$	→ $МНОЖЕСТВО_T$ {создается множество, элементы которого принадлежат либо одному, либо другому из исходных множеств}
{модификация}	пересечение: $МНОЖЕСТВО_T \times МНОЖЕСТВО_T$	→ $МНОЖЕСТВО_T$

Среди дополнительных свойств, связанных с этими операциями, можно отметить:

(E7) найти (выборка (e), e)
(E8) ¬найти (выборка–удаление$_T$ (e), выборка–удаление$_{МНОЖЕСТВОT}$ (e)) {два способа изображения индексов обозначают первую и вторую компоненты результата выборки–удаления}
(E9) найти (выборка–удаление$_T$ (e), e)
(E10) найти (t, объединение (e, e')) ⇒ найти (t, e) **или** найти (t, e')
(E11) найти (t, включить (e, e')) ⇒ найти (t, e) **и** найти (t, e') и т.д.

Все структуры, с которыми мы ниже познакомимся, представляют собой

частные случаи понятия множества, и операции создать–множество, включить, найти, убрать и т.д. мы еще встретим под различными другими названиями – по крайней мере некоторые из них, поскольку не все структуры в равной степени удобны для всех операций; так, операцию найти не легко реализовать на *стеке ;* с другой стороны, она составляет неотъемлемую часть спецификации *двоичных деревьев поиска.* Такие структуры, как *массивы,* не вполне подходят к операции убрать. Таким образом, выбор структур зависит от предполагаемых операций.

> Проблема работы с «множествами» будет затронута под другим углом зрения в VII.2 («Управление таблицами»). Часть материала несколько произвольно распределена между этим разделом и настоящей главой; например, таблицы «ассоциативной адресации», которые фигурируют в VII.2, могли бы быть описаны как структуры данных. Основная разница в том, что секция VII.2 делает упор на алгоритмы размещения и доступа и их характеристики («сложность»), а не на описания используемых структур.

Рассмотренные ниже структуры различаются главным образом способами, которыми в них выполняются последовательные «включения» и «удаления». В общем случае существует некоторое количество элементов, непосредственно доступных в каждый момент; доступ к другим элементам требует более сложных операций. Чем больше элементов доступно непосредственно, тем более богата структура; выигрыш в «богатстве» сопровождается, вообще говоря, потерями ясности, гибкости или простоты представления в памяти.

> Для пояснения структур данных часто заимствуют образную терминологию периферийных устройств ЭВМ. Так, можно уподоблять структуру некоторому «файлу»; включение и проверка найти аналогичны «записи» и «чтению» соответственно; могут существовать одна «читающая и записывающая головка» (стеки, массив) или две (файлы); доступ может быть «последовательным» (стеки, файлы) или «прямым» (массивы); запись может менять структуру (массивы) или не менять ее (стеки, файлы); то же самое относится к чтению. Важно подчеркнуть, что здесь эти термины *абсолютно не касаются физического представления;* они относятся к интуитивной функциональной спецификации структуры.

V.4. Стеки

V.4.1. Введение. Применения

Понятие *стека* часто встречается в повседневной жизни: кому не доводилось заподозрить свое начальство в том, что администратор складывает их бумаги в стек так, что пришедшие первыми документы лежат месяцами, прежде чем выберутся на поверхность? Именно эта характеристика – принцип «пришедший первым, уходит последним» – определяет такую структуру данных.

Стек – это структура с единственной читающей–записывающей головкой, последовательным доступом и неразрушающей записью. Более строго:

Стеком называется множество некоторого переменного (возможно, нулевого) числа данных, на котором выполняются следующие операции:

- пополнение стека новыми данными;
- проверка, определяющая, *пуст* ли стек;
- просмотр *последнего* прибавленного и не уничтоженного с тех пор данного, если такое существует;
- уничтожение последнего прибавленного и еще не уничтоженного данного, если оно есть.

Очевидна аналогия между стеком – этим важным понятием информатики – и обычной стопкой предметов, например книг: такая накапливающаяся на столе стопка представляет множество переменного числа книг; можно положить книгу в стопку, открыть книгу, расположенную на самом верху, или убрать ее из этой стопки. Но для того, чтобы посмотреть или извлечь другую книгу, надо прежде всего убрать все лежащие на ней книги.

В информатике стеки используются очень часто. Очевидным случаем является изучение физического процесса, в котором встречается настоящий стек. Рассмотрим пример с директором продовольственного магазина, который хочет получше организовать работу секции творожных сырков. Его разрывают два противоречивых желания: с одной стороны, сократить частоту заполнения секции и тем самым стоимость транспортно–погрузочных работ, а с другой стороны, уменьшить, насколько возможно, число коробок с сырками, на которые сверху положено слишком большое число других коробок, которые хранятся до предельной разрешенной для продажи даты, после чего их остается только выбросить. Чтобы лучше организовать работу секции, можно «промоделировать» последовательность добавлений и извлечений коробок с сырками. При этом надо попытаться «оптимизировать» последовательность добавлений; частота извлечения коробок в разные часы рабочего дня может быть приблизительно определена экспериментальными наблюдениями, при этом самая простая и самая разумная гипотеза состоит в том, что, за очень редкими исключениями, покупатель берет «коробку сверху». Таким образом, моделируется поведение одного или нескольких *стеков,* элементы которых характеризуются предельной датой продажи. Полагая приблизительно известной последовательность выборки коробок, надо, следовательно, определить идеальную последовательность пополнения, такую, что:

а) стек никогда не становится пустым;

б) элементы, находящиеся «на дне» стека, не остаются в стеке после предельной даты;

в) частота пополнения не слишком велика.

Разумеется, на практике необходимо прийти к компромиссу между этими целями.

Понятие стека вводится также при решении различных задач, относящихся собственно к информатике:

- *синтаксический анализ текста,* выполняемый в трансляторах языков высокого уровня, в значительной мере использует стеки. В частности, стеки постоянно применяются для распознавания «соотношений» между термами, которые следуют парами и могут быть вложенными, как скобки арифметических выражений или символы *BEGIN ... END* в программах АЛГОЛа W или ПЛ/1. Существует действительно прямое соответствие между поведением стека, где готовый к выборке элемент является всегда последним из посланных в стек элементов, структурой выражения, где каждая закрывающая скобка соответствует всегда последней встретившейся открывающей скобке, и между организацией программы блочной структуры, где каждый *END* «закрывает» последний встретившийся *BEGIN*.

  При трансляции программ блочной структуры можно заметить, что если в блоке объявлена новая переменная с тем же идентификатором, что и во включающем блоке, то новая переменная «маскирует» предыдущую: всюду в этом блоке новая переменная используется на месте соответствующей глобальной переменной, если только она сама не будет вытеснена омонимичной переменной, которая может встретиться во внутреннем блоке. Таким образом, омонимичные переменные образуют стек.

- выполнение рекурсивной подпрограммы также использует понятие стека. На самом деле, как мы увидим в гл. VI, последовательные «поколения» подпрограммы подчиняются «вложенной» схеме: каждое поколение может породить другие, но всякое поколение завершается до того, как управление возвращается на уровень «отцовского» поколения. Поэтому управление памятью совершенно естественно использует в этом случае стек. Некоторые ЭВМ имеют стековую архитектуру на базовом уровне (см., например, [IEEE 77]).

Пора ввести более точные определения.

V.4.2. Функциональная спецификация

Тип $СТЕК_T$, или *стек объектов типа* T, характеризуется операциями:

{создание}	создание стека: $\rightarrow СТЕК_T$ {функция без параметра, создающая пустой стек}
{доступ}	стекпуст: $СТЕК_T \rightarrow$ ЛОГИЧЕСКОЕ {проверка, является ли стек пустым}
{модификация}	засылка: $T \times СТЕК_T \rightarrow СТЕК_T$ {прибавление нового элемента типа T к стеку}
{модификация}	выборка: $СТЕК_T \rightarrow СТЕК_T$ {получение нового стека в результате извлечения одного элемента; свойства этой операции верны, только если стек не пуст}
{доступ)	последний: $СТЕК_T \rightarrow T$ {определение последнего прибавленного элемента)

Замечание: Две последние операции выборка и последний можно рассматривать единой операцией типа ($СТЕК_T \rightarrow (T \times СТЕК_T)$), результатом которой является элемент и новый стек. Соответствующие обозначения несколько тяжелее.

Перечисленные операции имеют следующие свойства для всякого t типа T и всякого с типа $СТЕКА_T$:

(C1)	стекпуст(созданиестека) = **истина** {создание стека создает пустой стек}
(C2)	стекпуст(засылка (t, c)) = **ложь** {если добавляется элемент к стеку, то результирующий стек не пуст}
(C3)	последний(засылка (t, c)) = c
(C'3)	последний (засылка (t, c)) = t {«восстановление» элементов в порядке, обратном тому, в котором они засылались в стек}
(C4)	~стекпуст (c) $\Rightarrow$ засылка (последний (c), последний (c)) = c {результатом выборки элемента из верхушки стека и последующего его возвращения является стек, идентичный исходному. Чтобы операция была определена, необходимо, чтобы исходный стек не был пуст}

Замечание: Для стеков, как и для других описываемых дальше структур, некоторые операции недопустимы, например выборка (c), если с пусто. На физическом уровне эти ситуации рассматриваются программами обработки ошибок. На уровне функциональных спецификаций условимся считать, что недопустимые операции, неявно запрещенные, приводят в результате к неиспользуемым объектам. Так, из посылки стекпуст(c), требуемой для C4, ничего нельзя доказать на недопустимом стеке.

Предоставим читателю проверку того, что эти свойства хорошо представляют интуитивную идею: «выборка элементов из стека осуществляется в порядке, обратном их засылке». Рассмотрим, например, выражение

E = последний (выборка (засылка (t_3, выборка (засылка (t_2, засылка (t_1c))))),

где t_1, t_2 и t_3 имеют тип T, а c – это произвольный стек типа $СТЕК_T$, пустой или непустой. Это выражение (читаемое справа налево) представляет последовательность операций: заслать t_1 в c; заслать t_2; выбрать элемент (в этом случае речь идет о t_2); заслать t_3 (который размещается над t_1); выбрать элемент (в этом случае речь идет о t_3); найти элемент верхушки стека; результатом является t_1. Это строго доказывается исходя из базовых свойств:

- по С3 выборка (засылка (t_2, засылка(t_1, c)) = засылка(t_1, c);
- в силу того же С3 засылка (выборка (t_3, засылка (t_1, c)) = засылка (t_1, c)
- и тогда по С3 E = последний (засылка (t_1, c)) == t_1

Точно так же по С3

засылка (t_1, последний (засылка (t_1, c))) = засылка (t_1, c)

В общем случае во всяком выражении такого вида, содержащем операции засылка и выборка, можно, двигаясь от правого края выражения, последовательно вычеркивать каждую операцию выборка и расположенную непосредственно справа от нее еще не вычеркнутую операцию засылка (если такой операции засылка не осталось, это означает, что выражение недопустимо, так как оно включает попытку выборки из пустого стека). Оставляем читателю поиск интуитивного смысла этого свойства, доказательство которого вытекает непосредственно из аксиомы С3.

V.4.3. Логическое описание

Самый «свежий» элемент стека, т.е. последний введенный и еще не уничтоженный, играет особенную роль: именно его можно рассмотреть или уничтожить. Этот элемент называется *верхушкой* стека. Оставшуюся часть можно назвать *телом* стека. Оно само является, по существу, стеком: если снять со стека его верхушку, то тело превращается в стек (Рис. V.13).

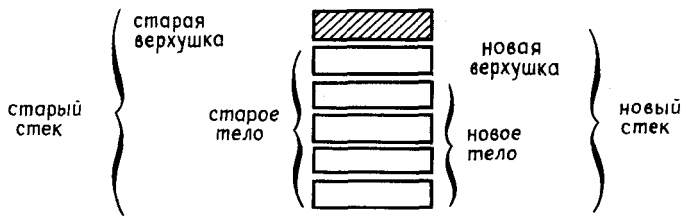


Рис. V.13

Поэтому естественное логическое описание состоит в рассмотрении стека как *соединения* элемента типа T, называемого «верхушкой», и некоторого стека. Чтобы получить конечные структуры, вводят (ср. V.1.3) возможность пустого стека

тип $СТЕК_T$= (ПУСТО | $НЕПУСТОЙСТЕК_т$)
тип $НЕПУСТОЙСТЕК_т$= (верхушка: T; тело: $СТЕК_T$)

где T, напомним, это тип объектов, засылаемых в стек: это может быть ЦЕЛОЕ, СТРОКА и т.д. или (почему бы нет?) $СТЕК_T$.

Легко реализовать операции функциональной спецификации:

программа созданиестека: $СТЕК_T$
  | созданиестека ← ПУСТО
программа стекпуст: ЛОГ **(аргумент** c: $СТЕК_T$)
  | стекпуст ← c **есть** ПУСТО
  | здесь конструкция **выбрать** ... неоправданно сложна и не используется}
программа засылка: $СТЕК_T$ **(аргументы** x: **T,** c: $СТЕК_T$)
  | засылка ← $СТЕК_T$ ($НЕПУСТОЙСТЕК_т$ (x,c))

```
программа выборка: СТЕК_Т (аргумент с: СТЕК_Т)
    | выборка ← тело (с)
программа последний : Т (аргумент с : СТЕК_Т)
    | последний ← если с есть ПУСТО то ошибка иначе тело (с)
```

V.4.4. Физическое представление

После того, что было показано в V.1.4, появляется мысль о двух возможных физических представлениях: при первом из них элементы расположены последовательно, при втором – размещены в произвольных местах, но связаны цепочкой указателей (Рис. V.14).

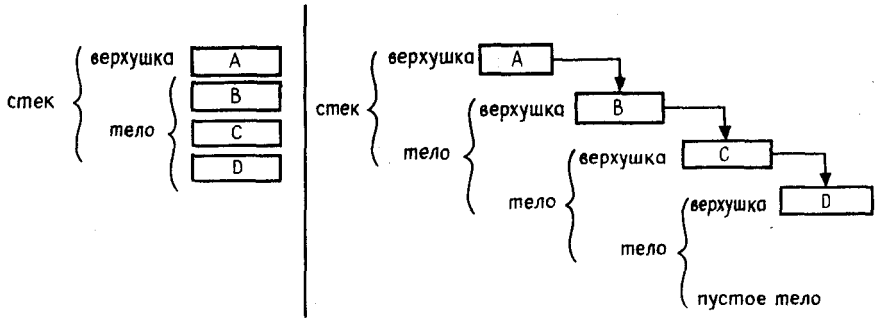


Рис. V.14 Сплошное представление, цепное представление.

ПЛ/1 предлагает, кроме того, третье простое представление, которого нет в других языках.

> Стоит отметить существование некоторых машин, архитектура которых была специально задумана для упрощения работы со стеками; можно назвать серию Burroughs 6700 [Органик 71], ICL 2900 [Бакл 76] и многие недавние «микропроцессоры». В таких машинах, предназначенных для того, чтобы облегчить выполнение программ, которые написаны на алголоподобных языках (блочная структура, рекурсия), память рассматривается скорее, как стек, чем как однородное множество слов; некоторые из базовых операторов являются операторами управления стеком. Все нижеследующее применяется в первую очередь к памяти, организованной более классическим способом.

V.4.4.1. Сплошное представление

Простое представление использует фиксированную область памяти и указатель; если a есть адрес первого элемента области, а c–адрес, отмечаемый этим указателем, то область, содержащая элементы стека в текущий момент, образуется элементами с адресами a, a + 1, ..., c. В языке высокого уровня можно использовать массив типа T, называемый, например, стек, и целый индекс, названный на Рис. V.15 индексом.

Пусть n – размер массива. Стек представляется с помощью:

```
| массив стек [1 : n] :Т
| переменная индекс: ЦЕЛОЕ
```

Подпрограмма созданиестека состоит в инициализации переменной индекс нулем. Другие подпрограммы записываются:

```
программа засылка (аргумент t:Т; модифицируемые параметры массив
                   стек[1:n]:Т, индекс: ЦЕЛ)
    если индекс = n то
        | ошибка
    иначе
        | индекс ← индекс + 1;
        | стек [индекс] ← t
```

```
программа выборка: Т (модифицируемые параметры массив стек [1 : n] : Т,
                      индекс : ЦЕЛ)
    если индекс = 0 то
        ошибка; выборка ← ПУСТО
    иначе
        выборка ← стек [индекс];
        индекс ← индекс – 1
программа стекпуст : ЛОГ (аргумент массив стек [1 : n] :Т, индекс: ЦЕЛ)
    стекпуст ← (индекс = 0)
```

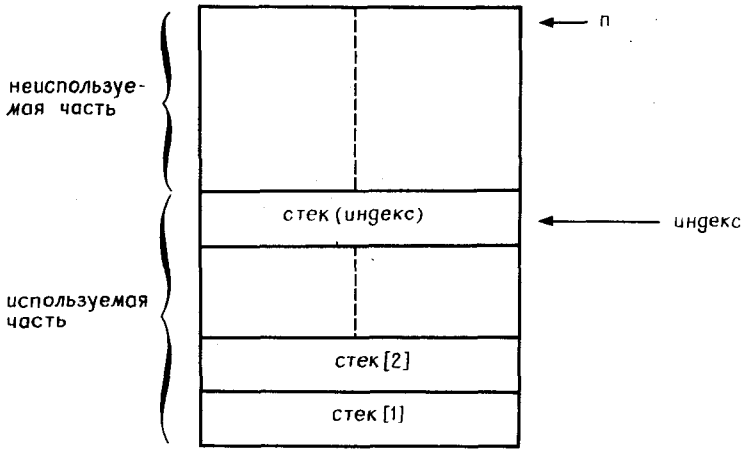


Рис. V.15 Сплошное представление стека.

Под «ошибкой» в вышеприведенных программах подразумевается включение процесса, позволяющего подсказать программисту, что была совершена ошибка, и решающего приостановить выполнение программы или вернуться к ней в приемлемых условиях: Существуют два возможных способа:

- выполнить вызов подпрограммы, которая печатает диагностическое сообщение, и не обрабатывать неверное предписание. Чтобы избежать неприятной путаницы, предпочтительнее не передавать в один и тот же файл нормальную выдачу программы и диагностику ошибок;
- присвоить значение ложь параметру ЛОГ типа результат, который мог бы называться «приемлемый запрос». Это обязывает постоянно передавать один дополнительный параметр.

И тот, и другой способы имеют свои неудобства, которые здесь не обсуждаются. Отметим, однако, разницу между логической ошибкой, которая может возникнуть в программе выборка (попытка выбрать из пустого стека), и ошибкой программы засылка, которая может появиться только на уровне физического представления и означать просто, что n было выбрано слишком малым.

Мы построили программу выборка такой, что она выдает значение выбранного элемента. Можно, разумеется, отдельно написать функцию последний:

```
программа последний (аргументы массив стек [1 :n] :Т,
                      индекс: ЦЕЛ)
    если индекс = 0 то
        ошибка;
        последний ← ПУСТО
    иначе последний ← стек [индекс]
```

Заметим, что операция выборка не разрушает физически элемент верхушки: в любой момент неиспользуемая часть стека может содержать данные, которые были помещены туда ранее, когда верхушка стека находилась «выше». Эти данные, в

принципе недоступные, будут стерты, только если размер стека увеличится снова. Тем не менее к ним **физически невозможно** обратиться: неверная работа с индексированной переменной вполне может вызвать обращение к данному, принадлежащему массиву; в стеке это невозможно. Наибольшее неудобство сплошного представления состоит с очевидностью в необходимости предусматривать заранее максимальный размер n, оценка которого всегда трудна Если n взять слишком большим, то резервируется неиспользуемая область памяти; если n слишком маленькое, то возможен преждевременный останов программы из–за переполнения (по.этому поводу отметим, что более целесообразно объявлять n как символическую константу или переменную, чем использовать в подпрограммах ее точное числовое значение: таким образом можно легко адаптироваться к размеру стека).

Интересная возможность появляется, когда надо управлять одновременно двумя стеками, и в частности, если исключена вероятность того, что оба стека становятся «полными» одновременно. Их представляют тогда *перевернутыми* в единственном массиве (Рис. V.16).

Стек B «изменяется в обратном направлении»: он пуст тогда и только тогда, когда индекс_B = n + 1; операция засылка *уменьшает* индекс_B на 1; операция выборка увеличивает его на 1. Когда пытаются заслать в тот и другой стек, переполнение имеет место тогда и только тогда, когда индекс_A = индекс_B.

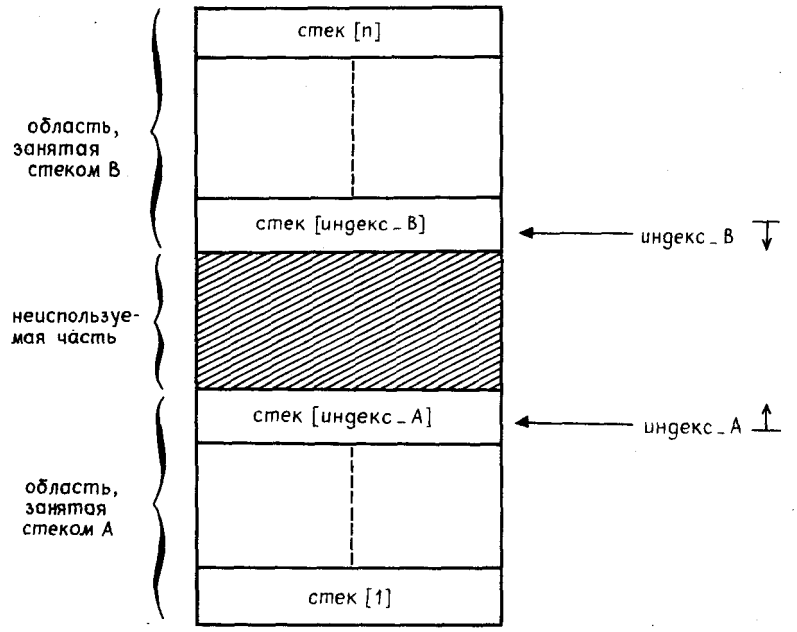


Рис. V.16 Представление двух стеков.

К сожалению, этот способ не обобщается больше чем на два стека.

Когда необходимо управлять m «параллельными» стеками, можно поставить им в соответствие m массивов. Если они имеют один и тот же тип, то их можно сгруппировать в один массив; в этом случае шагом изменения индекса будет m, а не 1.

V.4.4.2. Цепное представление

При цепном представлении каждый элемент стека состоит из значения и указателя, отмечающего предварительно засланный в стек элемент.

Цепное представление вызывает потерю места в памяти, связанную с наличием указателей, и представляет действительный интерес, только если, кроме представляемого стека, существуют другие структуры данных, для которых, с одной стороны, трудно определить максимальный размер, а с другой стороны, можно надеяться, что все эти структуры не достигают одновременно своих максимальных

размеров. Тогда свободное пространство памяти распределяется между ними способом «кучи» (V.1.4.2).

> Такая «куча» может в любом языке управляться программистом в виде массива, однако эти действия достаточно сложны, особенно если заниматься восстановлением освобождающихся областей; мы вернемся к этому в связи с линейными списками (V.6). Здесь мы ограничимся методами, используемыми в АЛГОЛе W, где эта работа поручена системе. ПЛ/1 предлагает эквивалентные возможности.

В АЛГОЛЕ W достаточно применить указывавшееся выше *логическое описание,* чтобы непосредственно получить цепное представление: предположим, что тип T определен объявлением

```
RECORD T(...)
```

тогда объявляется

```
COMMENT ПЕРЕВОДЫ ИМЕН PILE – СТЕК, TETE – ГОЛОВА, CORPS – ТЕЛО,
                CREER_PILE – СОЗДАНИЕ_СТЕКА, PILEVIDE – СТЕКПУСТ,
                DERNIER – ПОСЛЕДНИЙ EMPILER – ЗАСЫЛКА, DEPILER –
                ВЫБОРКА, ERREUR – ОШИБКА;
RECORD PILE–T (REFERENCE (T) TETE;
                REFERENCE (PILE_T) CORPS);
REFERENCE (PILE_T) PROCEDURE CREER_PILE;
                COMMENT ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ ЕСТЬ ЗНАЧЕНИЕ
                ИСПОЛЬЗУЕМОГО НИЖЕ ЛОГИЧЕСКОГО ВЫРАЖЕНИЯ;
                P = NULL;
REFERENCE (T) PROCEDURE DERNIER (REFERENCE (PILE_T)
                                        VALUE P); TETE (P);
PROCEDURE EMPILER (REFERENCE (T) X; REFERENCE
                                (PILE_T) VALUE RESULT P);
                P: = PILE_T (X,P);
PROCEDURE DEPILER (REFERENCE (PILE_T) VALUE RESULT P);
                IF PILEVIDE (P) THEN ERREUR
                COMMENT ВЫЗОВ ПОДПРОГРАММЫ ОБРАБОТКИ
                ОШИБОК;
                ELSE P : = CORPS (P)
```

Заметим, что засылка и выборка изменены по сравнению с функциональным определением: вместо возвращения значения типа стек (ссылка на стек) эти процедуры рассматривают свой параметр «стек» как модифицируемый параметр. Напомним, что, когда ссылка передается как **значение, результат** или **значение–результат,** при каждом вызове переписывается указатель, а не обозначаемая им структура данных.

Последние две процедуры можно было бы перегруппировать:

```
COMMENT RETRA1T – ВОЗВРАЩЕНИЕ
REFERENCE (T) PROCEDURE RETRAIT
                (REFERENCE (PILE_T) VALVE RESULT P);
        BEGIN REFERENCE (T) X;
        IF F = NULL THEN
            BEGIN
            ERREUR
            X := NULL
            END
        ELSE
            BEGIN
            X := TETE (P);
            P := CORPS (P)
            END
```

```
COMMENT СЛЕДУЮЩАЯ СТРОКА ЕСТЬ ВОЗВРАЩАЕМОЕ ЗНА-
ЧЕНИЕ; X
                END
```

До сих пор мы не уточняли, что такое тип T; было сказано только, что T – сложный тип, объявленный

```
RECORD T (...)
```

В действительности же, когда речь идет о стеках целых, вещественных или других простых типов, полезно сохранять эту формулировку. В самом деле, механизм *разделения вариантов* АЛГОЛа W позволяет записать

```
RECORD RI(INTEGER CONTENU);
RECORD RR(REAL CONTENU);
RECORD RS(STRING CONTENU);
RECORD T(REFERENCE (RI,RR,RS) ELEMENT)
```

и тогда можно использовать *те же самые* процедуры (что приводились выше), чтобы работать с разными стеками. Если в дальнейшем потребуется стек непредусмотренного типа, например *LONG COMPLEX,* достаточно изменить объявление T:

```
RECORD T (REFERENCE (RI, RR. RS, RLC) ELEMENT)
```

и объявить новую запись *RLC;* процедуры при этом останутся неизменными.

При этом способе возникает определенная неэффективность: засылается, например, не целое, а ссылка на ссылку на целое и система проверяет, что ссылки обозначают целое. Если единственный, используемый в программе стек является стеком целых, то процедуры доступа переписывают, заменяя *REFERENCE(T)* на *INTEGER.*

Однако в больших программах чаще используются многочисленные стеки различных типов, и тогда предлагаемый способ позволяет иметь *единый набор* процедур и, следовательно, значительно сократить возможность ошибок.

V.4.4.3. Конкретное представление стеков в ПЛ/1

Мы видели (секция V.2.2), что в ПЛ/1 существуют четыре способа распределения памяти, соответствующие четырем атрибутам – *STATIC, AUTOMATIC, CONTROLLED* и *BASED.* В частности, вспомним, что «фундаментальное свойство объектов, обладающих атрибутом *CONTROLLED,* состоит в том, что каждый оператор *ALLOCATE* создает новую версию этого объекта, которая маскирует предыдущую; оператор *FREE,* наоборот, уничтожает последнюю версию и обеспечивает доступ к предыдущей, если такая существует».

Этот механизм в точности соответствует механизму стека, в котором можно выполнять только прибавление верхушки или выборку верхушки и рассматривать только верхушку. Так же как стек может быть пустым, идентификатор с атрибутом *CONTROLLED* может в некоторый момент не обладать никаким действующим экземпляром либо потому, что не было предписаний *ALLOCATE,* либо потому, что их было столько же, сколько и операторов *FREE.* Проблема представления стека в ПЛ/1 решается поэтому очень просто. Стек можно объявить, записав, например,

```
DECLARE 1 PILE CONTROLLED,
          2   JOUR BINARY FIXED,
          2   MOIS BINARY FIXED,
          2   ANNEE BINARY FIXED;
/*JOUR – ДЕНЬ, MOIS – МЕСЯЦ, ANNEE – ГОД */
```

если, как в случае с коробками сырков, каждый элемент стека содержит дату в качестве данного. Стек становится пустым тогда и только тогда, когда *ALLOCATION(PILE)*

равно ложь (*'0'B);* в частности, этот случай имеет место перед первым исполняемым *ALLOCATE*.

Чтобы прибавить новую верхушку в стек, записывают

```
ALLOCATE PILE;
```

Если нужно дать значение содержимому этой новой верхушки, можно писать, например,

```
PILE. JOUR=19;
PILE. MOIS = 7;
PILE. ANNEE = 1980;
```

Наконец, для того чтобы убрать верхушку текущего стека, достаточно написать

```
FREE PILE;
```

Если необходимо обеспечить существование элемента в стеке, прежде чем пытаться его вызвать, надо написать

```
IF ALLOCATION (PILE) THEN FREE PILE;
```

V.4.5. Расширение понятия стека

Чтобы дополнить наше изучение стеков, следует добавить, что используемые на практике стеки не всегда строго подчиняются функциональной спецификации из V.4.2. Эта спецификация требует, чтобы добавление новых данных и их выборка всегда происходили в верхушке; без этого сам термин «стек» не имел бы большого смысла; но программа зачастую может посмотреть, читая без уничтожения, не только верхушку (функция последний), но и предшествующие элементы. В частности, так происходит при трансляции программ блочной структуры: в текущем блоке доступны не только локальные элементы этого блока, но и элементы включающего блока. Ясно, что на таком типе задач сплошное представление дает известные преимущества (Рис. V.17).

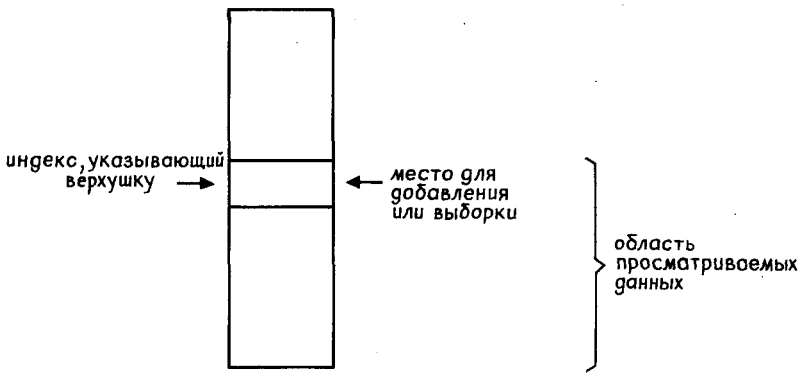


Рис. V.17 Представление расширенного стека.

V.5. Файлы

V.5.1. Введение. Применения

Очереди (или, еще короче, *файлы)* еще в большей степени, чем стеки, являются частью нашей повседневной жизни; по крайней мере мы это осознаем в большей степени, потому что нам ежедневно приходится быть частью такой структуры и даже требовать весьма страстно уважения основного принципа – «первый пришел – первый уходит».

Файл – это структура с одной читающей головкой и одной записывающей головкой, последовательным доступом и неразрушающей записью. Точнее:

Файлом называется множество переменного (возможно, нулевого) количества данных, на котором можно выполнять следующие операции:

- добавление нового данного;
- проверку, определяющую, *пуст* ли файл;
- просмотр *первого* записанного и не уничтоженного с тех пор данного (следовательно, самого давнего), если такое есть;
- уничтожение самого давнего данного.

Это определение хорошо согласуется с интуитивной концепцией очереди; если не учитывать безбилетников, перед окошечком кассы сначала обслуживается тот, кто прибыл первым и еще не обслужен, он и покидает очередь раньше.

Очереди имеют большое значение в информатике; они применяются к двум типам проблем:

- моделирование реальных очередей; современная техника связи прибавила к человеческим очередям целую гамму проблем, касающихся очередей сообщений, поступающих от «терминалов», которые связаны с одним или несколькими центрами связи. Стало обычным писать программы, исследующие поведение сетей для вычисления, например, максимального ожидания или среднего ожидания, моделируя реакции сети на случайные отправления сообщений. Таким же образом можно моделировать появление посетителей в банке (с тем чтобы определить количество окошек, открываемых в различные часы рабочего дня), очереди прибытия кораблей в порт и их отправления и т.д.
- решение задач собственно информатики, в частности в области операционных систем ЭВМ. Система имеет дело с целой серией запросов: начало выполнения работы, завершение работы, доступ к файлу, распределение памяти, печать результатов, выдача сообщений на пульт оператора и т.д. Некоторые типы запросов приоритетны по отношению к другим, но запросы одного типа должны удовлетворяться, вообще говоря, в порядке их поступления.

Надо подчеркнуть, что математическая статистика, способная дать некоторые теоретические результаты, касающиеся очередей, требует зачастую слишком упрощающих гипотез о способах, которыми реализуются входы и выходы. Практически же для получения более реалистических результатов приходится прибегать к моделированию.

V.5.2. Функциональная спецификация

Тип $ФАЙЛ_Т$, или *очередь объектов типа* Т, характеризуется операциями

создание файла:	$\rightarrow ФАЙЛ_Т$ {создание пустого файла}
файлпуст:	$ФАЙЛ_т \rightarrow$ ЛОГ {проверка пустоты файла}
дополнение:	$Т \times ФАЙЛ_Т \rightarrow ФАЙЛ_Т$ {прибавление одного элемента типа Т в файл}
удаление:	$ФАЙЛ_т \rightarrow ФАЙЛ_Т$ {получение нового файла путем удаления одного элемента; свойства этой операции верны, только если файл не пуст}
первый:	$ФАЙЛ_Т \rightarrow Т$ {доступ к самому давнему элементу файла, если файл не пуст}

Замечание: Операции первый и удаление могли бы быть объединены в одну операцию типа

($ФАЙЛ_Т \to Т \times ФАЙЛ_Т$), которая выполняет одновременно чтение и удаление первого элемента. Соответствующие обозначения выглядят тяжелее.

Пять описанных операций эффективно определяют очередь, если верны следующие четыре свойства (для всякого f типа $ФАЙЛ_Т$ и всякого t типа Т):

(Ф1) файлпуст (созданиефайла) = **истина** {созданиефайла создает пустой файл}

(Ф2) файлпуст (дополнение (t, f)) = **ложь** {если элемент включается в очередь, то результирующий файл не пуст}

(Ф3) первый (дополнение (t, f)) =
- t, если файлпуст (f)
- первый(0, если ~ файлпуст(f)

{элемент, включаемый в файл, становится в нем самым первым, только если файл был пуст, в противном случае первый элемент не изменяется}

(Ф'3) удаление (дополнение (t, f)) =
- создание файла, если файлпуст(f) {две последовательные операции – включение элемента в пустой файл и удаление – оставляют файл пустым}
- дополнение (t, удаление (f)), если ~файл(f) {две последовательные операции – включение элемента в непустой файл и удаление – коммутативны: их порядок несуществен}

Сравним эти операции и свойства с теми, которые были определены в V.4.2 для стеков. Заметно соответствие операций:

созданиестека	и	созданиефайла
стекпуст	и	файлпуст
засылка	и	дополнение
выборка	и	удаление
последний	и	первый

В силу этих «соответствий» свойства С1 и С2 эквивалентны Ф1 и Ф2; Различие двух структур определяется различием С3 и Ф3, С'3 и Ф'3 (эквивалент С4 верен для файлов). Заметна роль, которую играют абстрактные свойства, определенные на этих структурах.

Свойства Ф1 и Ф3 дают три следствия, два из которых особенно отличаются от соответствующих свойств для стеков.

(Ф5) файлпуст (f) $\Rightarrow$ файлпуст (удаление (дополнение (t, f)))

Это свойство выражает просто первую половину Ф'3. Оно означает: если в пустой файл включается элемент и непосредственно сразу же удаляется, то файл вновь будет пустым (это, признаем, разумно).

(Ф6) если файлпуст (f), то имеет место эквивалентность: дополнение (первый (f), удаление (f)) = f $\Leftrightarrow$
файлпуст (удаление (f))

(т.е., извлекая объект из файла и непосредственно сразу же возвращая его в файл, получают файл, равный исходному в том и только том случае, когда в файле был только один элемент).

а) это свойство верно, если удаление (f) пусто

б) две функции модификации дополнение и удаление оставляют это свойство неизменным. Например, если f не пуст:

дополнение (первый (дополнение (t, f)), удаление (дополнение (t, f)))
Ф3.Ф'3
=== дополнение (первый (f), удаление (f))

(Ф7) если к изначально пустому файлу применяются m дополнений и m удалений в произвольном порядке, но так, что каждая операция правомерна (т.е. число выполненных удалений не превосходит числа добавлений), то m удаленных элементов–это m добавленных элементов в том же порядке.

Это, конечно, главная характеристика очереди. Заметим, однако, что файл должен быть пустым вначале, что не требуется для соответствующего свойства стека, зато порядок операций строго не фиксируется.

Выраженное в терминах операций функциональной спецификации следствие Ф7 означает, что если E – выражение, полученное применением к пустому файлу дополнение и удаление, и если E «правомерно» (т.е. при просмотре выражения справа всегда встретится столько же дополнений, сколько и удалений), то из E можно, двигаясь справа, вычеркнуть все операции удаление и столько же операций дополнение.

Пример: если f пуст и если

E = дополнение (t4, удаление (дополнение (t3, удаление (удаление (дополнение (t2, дополнение (tl, f)))))))

то

E = дополнение (t4, f)

и первый (E) = t4

Если f не пуст, то это свойство неверно, но всегда можно, исходя из Ф'3, восстановить все операции удаления слева

E = удаление (удаление (дополнение (4, дополнение (t3, дополнение (t2, дополнение (tl,f)))))))

V.5.3. Логическое описание

Интутитивное определение и функциональная спецификация изображают файл в виде последовательности объектов, в которой два крайних играют особую роль: *голова* файла–это объект, получаемый операцией «первый» и уничтожаемый операцией «удаление»; *хвост* –это последний объект, введенный операцией «дополнение» (Рис. V.18).

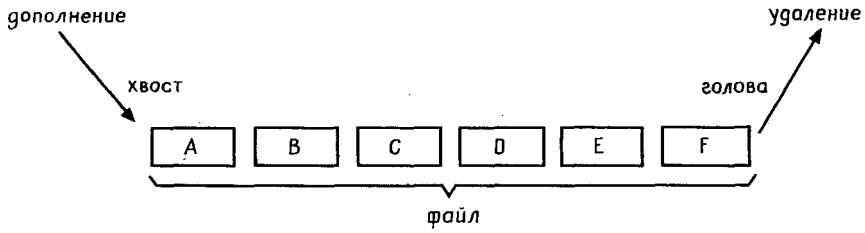


Рис. V.18 Файл.

Требование использовать эту структуру «с двух концов» приводит к тому, что рекурсивное описание структуры должно сопровождаться рекурсивным описанием программ[1].

Действительно, если написать

тип ФАЙЛ$_T$ = (ПУСТО | НЕПУСТОЙФАЙЛ$_T$)
тип НЕПУСТОЙФАЙЛ, = (голова: T; тело: ФАЙЛ$_T$)

где T – тип объектов, составляющих файл, то легко реализовать четыре из пяти

[1] В подпрограмме, содержащей вызов самой себя (рекурсивной подпрограмме), такой вызов обозначен *курсивом*. Рекурсивные программы изучаются в следующей главе.

операций:

программа созданиефайла: ФАЙЛ$_T$
| созданиефайла ← ПУСТО
программа файлпуст: **ЛОГ (аргумент:** ФАЙЛ$_T$)
| файлпуст ← f **есть** ПУСТО
программа первый: T **(аргумент** f: ФАЙЛ$_T$)
| **если** i **есть** ПУСТО **то** ошибка **иначе** первый ← голова (f)
программа удаление: ФАЙЛ$_T$ (аргумент :ФАЙЛ$_T$)
| **если есть** ПУСТО **то** ошибка **иначе** удаление ← тело (f)

Подпрограмма дополнение должна прибавлять элемент в «хвост» файла, к которому нет непосредственного доступа. Тем не менее можно различать случаи:

- f есть ПУСТО : тогда дополнение (t, f) может быть получено формированием файла НЕПУСТОЙФАЙЛ(t, f), так как голова и хвост совпадают в файле из одного элемента;
- в противном случае, т.е., если f есть НЕПУСТОЙФАЙЛ, то «дополнение» файла f элементом t означает дополнение файла тело(f). Предваряя рекурсию, являющуюся предметом следующей главы, можно, следовательно, определить рекурсивно подпрограмму дополнение совершенно законным образом:

программа: дополнение: ФАЙЛ$_T$ (аргументы t: T, f :ФАЙЛ$_T$)
| **если** f есть ПУСТО то дополнение ← НЕПУСТОЙФАЙЛ (t, f)
| **иначе** дополнение ← НЕПУСТОЙФАЙЛ (первый (f). *дополнение*
(t, удаление (f)))

V.5.4. Физическое представление

Последняя подпрограмма иллюстрирует практически важный недостаток нашего логического описания: оно не дает непосредственного доступа к *хвосту* файла, что упрощало бы операцию дополнение.

Как при сплошном, так и при цепном представлении в файле задают две точки входа (Рис. V.19).

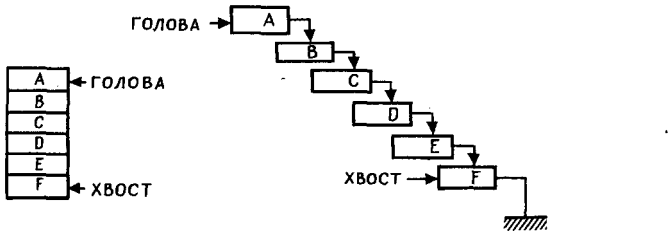


Рис. V.19 Сплошное и цепное представление очереди.

Как и раньше, мы воспользуемся **АЛГОЛом W** для иллюстрации цепного представления при динамическом распределении и **ФОРТРАНОМ** для сплошного представления при статическом распределении. ПЛ/1 не предлагает здесь особых свойств.

V.5.4.1. Цепное представление

Предположив, что тип T определен объявлением *RECORD T(...)*
можно объявить

RECORD FILE_T(REFERENCE(T) TETE; REFERENCE (FILE_T) CORPS);
COMMENT: TETE – ГОЛОВА, CORPS – ТЕЛО;

так же как и две точки входа каждой очереди, используемой в программе

```
REFERENCE (FILE_T )TETE,QUEUE; COMMENT: QUEUE– ХВОСТ;
```

Отметим, что *F1LE_T* вполне описывает ФАЙЛ, а не только НЕПУСТОЙФАЙЛ, поскольку ссылка на файл всегда может иметь значение *NULL*

Тогда подпрограммы описываются:

```
COMMENT: ПЕРЕВОДЫ ИМЕН ПРОЦЕДУР
        CREERFILE–СОЗДАНИЕФАЙЛА, FILEVIDE–ФАЙЛПУСТ,
        PREMIER – ПЕРВЫЙ, DEFILER – УДАЛЕНИЕ,
        ENFILER – ДОПОЛНЕНИЕ;
PROCEDURE CREERFILE (REFERENCE (FILE_T) RESULT TF, QF);
COMMENT :TF И QF – ГОЛОВА И ХВОСТ ФАЙЛА;
        TF : = QF : = NULL
        LOGICAL PROCEDURE FILEVIDE (REFERENCE (FILE_T) VALUE TF);
                COMMENT: РЕЗУЛЬТАТ ЗАДАЕТСЯ ЛОГИЧЕСКИМ
                        ВЫРАЖЕНИЕМ;
        TF = NULL;
            ;
REFERENCE(T) PROCEDURE PREMIER (REFERENCE (FILE_T) VALUE TF);
        IF TF = NULL THEN ERREUR1 ELSE TETE(TF);
PROCEDURE DEFILER (REFERENCE (FILE–T) VALUE RESULT TF);
        IF TF = NULL THEN ERREUR2 ELSE TF: = CORPS (TF);
PROCEDURE EN FILER (REFERENCE (T) VALUE X;
        REFERENCE (FILE_T) VALUE RESULT TF, QF);
        COMMENT: ГОЛОВА ФАЙЛА ИЗМЕНЯЕТСЯ, ЕСЛИ ФАЙЛ
        БЫЛ ПУСТЫМ. В ПРОТИВНОМ СЛУЧАЕ ИЗМЕНЯЕТСЯ
        ТОЛЬКО ХВОСТ. РЕКУРСИВНЫЙ АЛГОРИТМ БЫЛ БЫ В
        РАВНОЙ СТЕПЕНИ ВОЗМОЖЕН КАК В АЛГОЛЕ W, ТАК
        И В ПЛ/1;
        IF TF = NULL THEN TF : = QF : = FILE_T(X, NULL)
        ELSE BEGIN
            CORPS (QF) : = FILE_T (X.NULL);
            QF : = CORPS (QF)
            COMMENT НУЖНО ПЕРЕМЕСТИТЬ ХВОСТ;
            END
```

Отметим тонкость в написании *ENFILER* : не надо забывать изменять *QF* после прибавления *X* (Рис. V.20).

В программах, работающих с очередями объектов одного простого типа, например *REAL*, всюду заменяют на этот тип ссылку *REFERENCE (T)*.

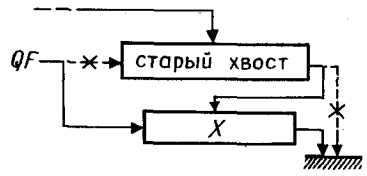
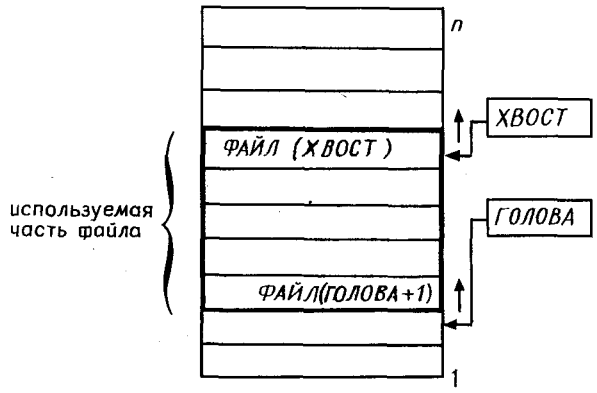


Рис. V.20

V.5.4.2. Сплошное представление

В том случае, когда цепные связи между последовательными элементами очереди требуется заменить простым отношением смежности в памяти, используется одномерный массив *ФАЙЛ* (в ФОРТРАНе, вообще говоря, несколько параллельных массивов, если только объекты не принадлежат одному и тому же простому типу) и

устанавливаются два индекса, отмечающих соответственно голову и хвост файла. Файл имеет максимальный размер, равный объявленной длине массива, и схематически изображается следующим образом:



Индекс *ГОЛОВА* указывает позицию, за которой следует голова файла, т.е. ту позицию, откуда делалось последнее удаление. Это соглашение далее будет оправдано

Ставится задача: даже если размер файла всегда остается меньше разрешенного максимума *m* файл неумолимо «поднимается», так как удаление выполняется снизу, а дополнения – сверху. Если не принять мер предосторожности, файл переполнит массив после *n* операций дополнение.

Возможны несколько решений:

- при каждом удалении восстанавливать освобожденное внизу массива место, «опуская» весь файл на одну ступеньку. Такое решение реализовать просто, но для больших файлов оно является дорогостоящим;
- позволить файлу подниматься до тех пор, пока остается место для выполнения дополнений; когда места больше нет и надо выполнять очередное дополнение, восстановить сразу все пространство, освобожденное удалениями, «спуская» файл на всю возможную высоту. Это решение более экономично, но все же требует бесполезных перемещений информации;
- когда хвост достигает вершины массива, выполнять последующие дополнения в файл снизу массива, как показано на Рис. V.21.

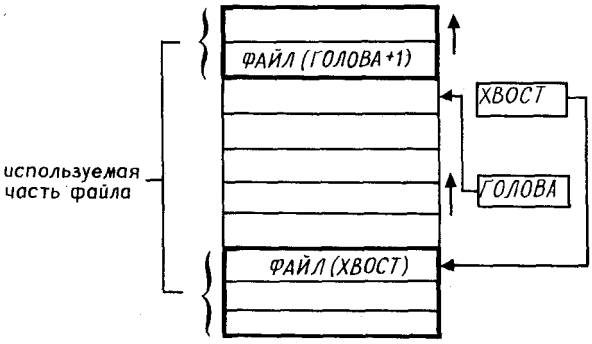


Рис. V.21

Таким образом, индексы *ГОЛОВА* и *ХВОСТ* рассматриваются как целые индексы по модулю n, размеру массива. Ценность этого метода состоит в том, что он не требует никакого дополнительного места в памяти; такое представление называют *циклическим файлом;* два предыдущих рисунка можно изобразить в следующем виде (Рис. V.22):

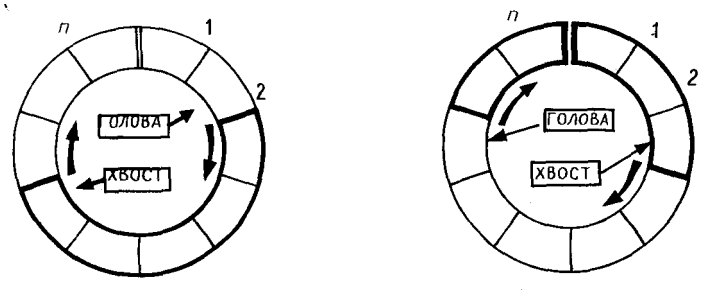


Рис. V.22 Циклические файлы.

Программирование такого решения представляет собой ловушку: проверка, определяющая, пуст ли файл, теперь записывается *ГОЛОВА = ХВОСТ;* но равенство *ГОЛОВА = ХВОСТ* имеет место также, если файл содержит *n* элементов. Чтобы различать эти два случая (файл пустой и файл полный), максимальным размером файла считают *n — 1* (а не *n*). При таком условии

$|ГОЛОВА - ХВОСТ| \geq 1$. если файл не пуст.

Заметим, что сделанный выбор индексов *ГОЛОВА* и *ХВОСТ* с *ГОЛОВОЙ,* отмечающей позицию, пройденную головой файла, не оказывает влияния на саму задачу; это соглашение только позволяет упростить написание функций файлпуст и дополнение.

При таком ограничении файл максимального размера 100 в ФОРТРАНе объявляется путем

```
REAL FILE (101)
INTEGER TETE, QUEUE, MAXI
DATA MAXI /101/
```

Создание файла описывается в виде

ФОРТРАН

```
SUBROUTINE CREFIL (TETE, QUEUE)
INTEGER TETE, QUEUE
TETE = 1
QUEUE = 1
RETURN
END
```

Единица допустима здесь так же, как и любое значение, заключенное между 1 и *MAXI,* так как равенство *TETE* и *OUEUE* означает пустоту файла. Функция *FILEVID* (*ФАЙЛПУСТ*) представляется просто отношением: (*TETE.EQ.QUEUE).*

Другие подпрограммы имеют вид

ФОРТРАН

```
      REAL FUNCTION PREM(FILE, TETE, QUEUE, MAXI)
C     PREM– ПЕРВЫЙ
      REAL FILE (MAXI)
      INTEGER TETE, QUEUE, MAXI
      IF(TETE.EQ.QUEUE) CALL ERREUR
      I = TETE + 1
      IF(I.GT.MAXI)I = 1
      PREM = FILE (I)
      RETURN
      END

      SUBROUTINE DEFIL (FILE, TETE, QUEUE, MAXI)
C     DEFIL–УДАЛЕНИЕ
      REAL FILE (MAXI)
      INTEGER TETE, QUEUE, MAXI
      IF (TETE.EQ.QUEUE) CALL ERREUR
      TETE = TETE + 1
      IF(TETE.GT.MAXI) TETE=1
      RETURN
      END
```

Предпоследний оператор завершает возможный цикл, возвращает в начало массива. Подпрограммы *PREM* и *DEFIL* часто объединяются в единую программу (неразрушающее чтение):

ФОРТРАН

```
      SUBROUTINE DEFPRE (FILE, TETE, QUEUE, MAXI, PREM)
C     DEFPRE – УДАЛЕНИЕ_ПЕРВЫЙ
      REAL FILE (MAXI)
      INTEGER TETE, QUEUE, MAXI, PREM
      IF (TETE.EQ.QUEUE) CALL ERREUR
      TETE = TETE + 1
      IF (TETE.GT.MAXI) TETE = 1
      PREM = FILE (TETE)
      RETURN
      END
```

Наконец, дополнение файла элементом *X* выполняется так:

ФОРТРАН

```
      SUBROUTINE ENFIL (FILE, TETE, QUEUE, MAXI,X)
C     ENFIL – ДОПОЛНЕНИЕ
      REAL FILE(MAXI),X
      INTEGER TETE, QUEUE,MAXI
      QUEUE = QUEUE + 1
      IF (QUEUE.GT.MAXI) QUEUE = 1
      IF (TETE.EQ.QUEUE) CALL ERREUR
      FILE(QUEUE) = X
      RETURN
      END
```

После увеличения *QUEUE* равенство *QUEUE = TETE* не может больше означать пустоту файла, наоборот, это свидетельствует о том, что файл заполняет весь массив, что запрещено; отсюда–вид последнего условного оператора.

V.5.5. Обобщение: файл с двойным доступом

Файлы с **двойным доступом** являются обобщением предыдущих структур. *Стек* доступен только с одного конца: можно прибавлять новый элемент и рассматривать или уничтожать последний прибавленный элемент. *Файл* доступен с одного конца для прибавления нового элемента, а с другого конца–для рассмотрения или удаления первого из прибавленных, но еще не уничтоженных элементов (Рис. V.23).

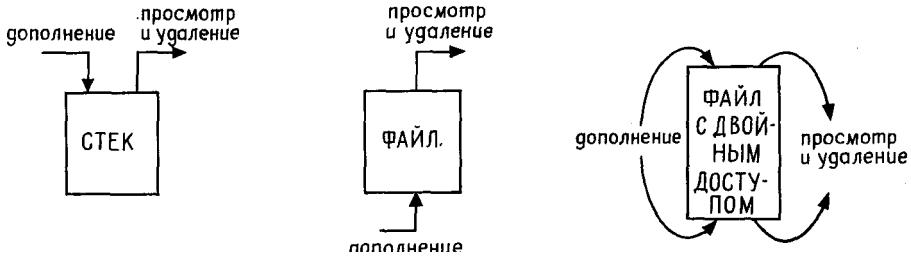


Рис. V.23 Файл с двойным доступом.

Файл с двойным доступом объединяет эти свойства, разрешая прибавление, просмотр и удаление с обоих концов. Читатель может при желании написать полную функциональную спецификацию, уточняющую это интуитивное определение.

Следует заметить, что файлы с двойным доступом редко встречаются во всей их общности; чаще имеют дело с файлами, для которых только дополнение или только удаление выполняется с двух концов, другая операция выполняется только с одного конца. Случай, когда только удаление выполняется с одного конца, это ситуация очереди, где некоторые «пользователи» имеют приоритеты перед другими.

Логическое описание файла с двойным доступом должно предусматривать раздвоение операций дополнения, просмотра и удаления, учитывающее существование двух «концов». Оно допускает сплошное представление, подобное представлению циклических файлов или «двойным связям», которые будут введены ниже в связи с линейными списками.

V.6. Линейные списки

V.6.1. Введение. Применения

Линейные списки, к изучению которых мы сейчас приступаем, являются новым обобщением предыдущих структур; они позволят нам впервые представить множество (V.3) так, чтобы каждый элемент был доступен и при этом не нужно было бы извлекать некоторые другие.

Будем придерживаться терминологии, введенной Кнутом, называя «линейными списками» эти структуры и сохраняя название «списки» для структур, более сложных и фактически древовидных. Такие структуры встретятся в V.8. Отметим, однако, что некоторая путаница в применении этих двух терминов все же существует.

Линейный список – это предлагаемое информатикой представление конечного и упорядоченного множества элементов типа. Математически он может обозначаться простым перечислением своих элементов в заданном порядке; например,

$$\ell = (t_1, t_2, t_3, t_4, t_5)$$

Таким образом, линейные списки будут естественно использоваться всякий раз, когда встречаются упорядоченные множества переменного размера, где операции

включения, поиска, удаления и т.д. должны выполняться не систематически в голове или хвосте, как для файлов или стеков, а в произвольных местах, но с сохранением порядка. Таким порядком могли бы быть, например, *приоритеты,* присвоенные заданиям, ожидающим обработки в операционной системе: достаточно файла, если применяемый принцип прост – «первый пришел – первым обслуживается», но необходим линейный список, если стратегия менее примитивна.

В *анализе* текста, используемом в информатике, например при трансляции (*синтаксический анализ),* встречаются достаточно часто линейные списки. Таковы следующие «фразы» АЛГОЛа W:

INTEGER I,J,K,L,M;
RECORDR(INTEGER A; REAL B; LONG REAL C)

V.6.2. Функциональная спецификация

Функциональная спецификация линейного списка объектов типа T, или $ЛС_T$, которую мы подробно не приводим, так как она достаточно длинна, включает функции:

следующий: $T \times ЛС_T \to T$ {следующий(t, *ℓℓ*)–это объект, следующий за t в списке *ℓℓ,* если такой объект существует}
вставка: $T \times T \times ЛС_T \to ЛС_T$ {вставка (t, t', *ℓℓ)* дает новый список, в котором t вставлено перед t' или после всех элементов, если t' не принадлежит *ℓℓ*}
исключение: $T \times ЛС_T \to ЛС_T$ {удалить элемент из списка}
первый: $ЛС_T \to T$ {дает первый элемент, если он существует; ср. «последний» для стеков, «первый» для файлов}

V.6.3. Логическое описание

Линейный список является последовательностью объектов. Поэтому логическое описание включает, так же как для стеков и очередей, объявления вида

тип ЛИНЕЙНЫЙСПИСОК$_т$=(ПУСТО | НЕПУСТОЙЛИНСПИСОК$_т$)
тип НЕПУСТОЙЛИНСПИСОК$_т$= (начало: T;
продолжение: ЛИНЕЙНЫЙСПИСОК$_т$)

где T – тип элементов, составляющих линейный список. Тогда можно рекурсивно описать операции функциональной спецификации:

программа созданиесписка: ЛИНЕЙНЫЙСПИСОК$_т$
| созданиесписка ← ПУСТО
программа списокпуст: ЛОГ (аргумент *ℓℓ:* ЛИНЕЙНЫЙСПИСОК$_т$)
| списокпуст ← *ℓℓ* **есть** ПУСТО
программа первый: **T (аргумент** *ℓℓ:* НЕПУСТОЙЛИНСПИСОК$_т$)
| первый ← начало (*ℓℓ)*
программа вставка: ЛИНЕЙНЫЙСПИСОК$_т$ **(аргументы** t, t':T,
ℓℓ : ЛИНЕЙНЫЙСПИСОК$_т$)
| {рекурсивная программа}
| вставка ← **если** *ℓℓ* **есть** ПУСТО **то**
| НЕПУСТОЙЛИНСПИСОК$_т$(t, ПУСТО)
| **иначе если** первый *(ℓℓ)* = t' **то**
| НЕПУСТОЙЛИНСПИСОК$_т$ (t, *ℓℓ)*
| **иначе**
| НЕПУСТОЙЛИНСПИСОКт
| (начало (*ℓℓ*), *вставка* **(**t, t', продолжение(*ℓℓ*))**)**

```
программа исключение: ЛИНЕЙНЫЙСПИСОКт (аргументы t : T,
                                        ℓℓ : ЛИНЕЙНЫЙСПИСОКт)
    {рекурсивная программа}
    исключение ← если ℓℓ есть ПУСТО то
    иначе если начало(ℓℓ) = t то продолжение(ℓℓ)
    иначе НЕПУСТОЙЛИНСПИСОКт
    (начало (ℓℓ),
    исключение(t, продолжение (ℓℓ)))
```

Нерекурсивные версии будут даны ниже.

V.6.4. Физические представления

Как и для ранее рассмотренных линейных структур, существуют возможности сплошного представления массива и цепного представления. Но на этот раз при сплошном представлении никаким способом не удается избежать физического *перемещения* некоторых элементов в реализации вставки. Действительно, чтобы перейти от

A	B	C	D	E

к

A	B	C	X	D	E

путём включения нового элемента X, необходимо изменить места по крайней мере двух элементов D и E. Точно так же удаление обязывает уплотнить список, чтобы «поглотить дыру», оставленную удаляемым элементом. Это решение по мере увеличения длины списков быстро становится неэффективным во время выполнения. Поэтому даже в таком мало приспособленном языке, как ФОРТРАН, предпочтительнее цепное представление, где включение элемента состоит в установке двух указателей. Данные при этом не перемещаются (Рис. V.24).

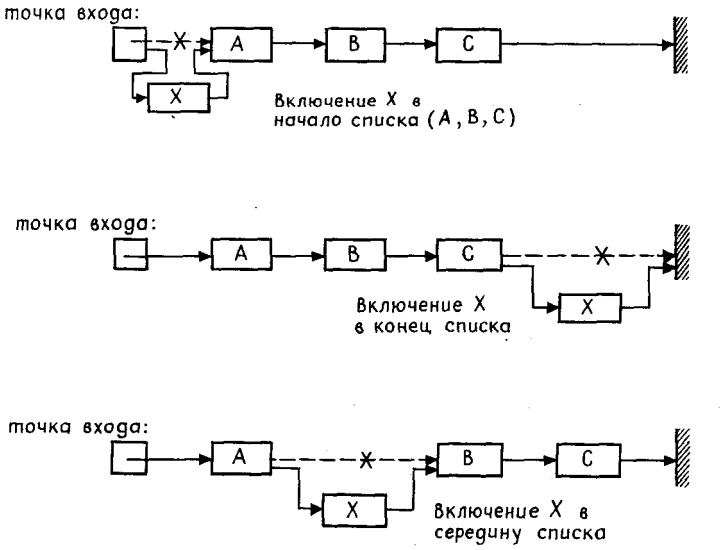


Рис. V.24 Включение элемента в линейный список.

Эти же схемы позволяют увидеть, как происходит удаление элемента; на этот раз достаточно изменить только один указатель, отмечающий удаляемый элемент; тот факт, что указатель, исходящий из удаленного» элемента, продолжает существовать, не является ограничением, так как доступ к этому элементу прекращен.

Система указателей, связывающая различные элементы списка, легко вписывается в АЛГОЛ W с помощью ссылок *REFERENCE* и в ПЛ/1 с помощью указателей *POINTER*. Представление линейных списков в этих языках выводится поэтому очень просто из логического описания. Единственной проблемой является

замена рекурсивных вызовов на итерации, если нужны более эффективные алгоритмы. Если в ПЛ/1 объявлено

```
DECLARE 1 LIST E_LINE AIRE BASED (P),
                  2 DEBUT POINTER,
                  2 SUITE POINTER;
/* L1STE_LINEAIRE – ЛИНЕЙНЫЙ.СПИСОК, DEBUT – НАЧАЛО,
                                          SUITE – ПРОДОЛЖЕНИЕ */
```

и если, кроме того, предполагается, что тип элементов списка определен некоторой другой структурой, то подпрограммы вставки и исключения можно записать в нерекурсивной форме. Для упрощения рассмотрен случай списка строк:

ПЛ/1

```
/* ПЕРЕВОДЫ ИМЕН: ELEML1STE – ЭЛЕМЕНТ–СПИСКА, ELEMNOUV – НОВЫЙ
                        ЭЛЕМЕНТ, LIS – СПИС, INSERAV – ВСТАВКА */
INSERAV : PROCEDURE (ELEMNOUV, ELEMUSTE, LIS),
      /* ВСТАВИТЬ НОВЫЙ ЭЛЕМЕНТ ПЕРЕД ЭЛЕМЕНТОМ СПИСКА В СПИСКЕ
         СПИС ИЛИ В КОНЦЕ СПИСКА, ЕСЛИ ЭЛЕМЕНТ.СПИСКА
         ОТСУТСТВУЕТ В СПИС */
 DECLARE (ELEMNOUV, ELEMLISTE) CHARACTER (20) VARYING LIS POINTER;
/* "ПОДГОТОВИТЬ" НОВУЮ ЯЧЕЙКУ К ВКЛЮЧЕНИЮ*/
ALLOCATE LISTE_LINEAIRE;
P –> LISTE_LINEAIRE.DEBUT = ELEMNOUV;
IF LIS = NULL THEN
      DO
      LIS = P; LIS –> LISTE_LINEAIRE.SUITE = NULL THEN
      END;
ELSE
      BEGIN
/* ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ ДЛЯ ПРОХОЖДЕНИЯ СПИСКА */
/* TROUVE–ОБНАРУЖЕН */
DECLARE L POINTER, TROUVE BIT (1);
       L = LIS; TROUVE = '0'B;
       DO WHILE ¬TROUVE;
            IF L –> LISTE–LINEAIRE.SUITE = NULL THEN
            TROUVE = '1'B;
            ELSE IF L–> LISTE_LINEAIRE. SUITE –> L1STE_LINEA1RE.DEBUT
                  = ELEMLISTE THEN
                  TROUVE = '1'B;
            ELSE L = L–> LISTE_LINEAIRE.SUITE;
            END;
      /* ЗДЕСЬ L ОЗНАЧАЕТ ЛИБО ПРЕДШЕСТВУЮЩИЙ ЭЛЕМЕНТ СПИСКА,
         ЛИБО ПОСЛЕДНИЙ, ЕСЛИ ЭЛЕМЕНТ СПИСКА НЕ ПРИСУТСТВУЕТ В
         СПИСКЕ */
      /* УСТАНОВКА УКАЗАТЕЛЕЙ ДЛЯ ВКЛЮЧЕНИЯ */
  P –> LISTE_LINEAIRE.SUITE = L –> LISTE_L1NEAIRE.SUITE;
  L –> LISTE–LINEAIRE.SUITE = P
  END;
  END INSERAV;
```

Цикл *WHILE* в этой подпрограмме не достаточно изящен. Его следовало бы написать проще:

пока ℓ.продолжение ≠ null и ℓ.продолжение ≠ элемент_списка
повторять
| ℓ ← ℓ.продолжение

Однако такая нотация возможна, только если в операции и ее второй элемент не вычисляется, когда первый имеет значение **ложь** (*ℓ.продолжение = NULL)* Это свойство обеспечено в АЛГОЛе W, но не в ПЛ/1, который обязывает ввести переменную *TPOUVE* и писать более тяжеловесно, чтобы избежать незаконного вычисления ℓ.продолжение ← ℓ.начало когда *ℓ. продолжение = null.*

Подобное же замечание относится и к подпрограмме *DETRUIRE* (*ИСКЛЮЧЕНИЕ*). В обоих случаях заметьте, насколько с точки зрения логического описания рекурсивная форма более громоздка, чем нерекурсивная.

ПЛ/1

```
/* ELEM – ЭЛЕМ, PRECEDENT – ПРЕДЫДУЩИЙ */
DETRUIRE: PROCEDURE (ELEM, LIS);
/*ИСКЛЮЧИТЬ ЭЛЕМЕНТ ЭЛЕМ ИЗ СПИСКА СПИС. ЕСЛИ Н ТАМ ЕСТЬ*/
DECLARE ELEM CHARACTER(100) VARYING,
LIS POINTER;
DECLARE (L, PRECEDENT) POINTER;
  IF LIS ¬= NULL THEN
  DO;
      IF LIS –> LISTE.LINEAIRE.DEBUT = ELEM THEN
            LIS = LIS–>LISTE–LINEAIRE.SUITE;
      ELSE
          BEGIN;
          /* ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ ДЛЯ ПРОХОЖДЕНИЯ СПИСКА*/
          DECLARE L POINTER, TROUVE BIT (I);
          L = LIS; TROUVE = '0' B;
          DO WHILE ¬TROUVE;
              IF L–> LISTE_LINEAIRE. SUITE = NULL
              THEN TROUVE = T B;
              ELSE IF L –> LISTE_LINEA1RE.SUITE
              –> LISTE_LINEAIRE.DEBUT = ELEM
              THEN DO;
                  /* ИЗМЕНЕНИЕ СПИСКА */
                  L –> LISTE_LINEAIRE.SUITE =
                  L –> LISTE_LINEAIRE.DEBUT;
                  TROUVE = '1'B;
                  END;
              ELSE L = L –> LISTE_LINEAIRE.SUITE;
              END;
          END;
      END;
  END DETRUIRE;
```

В противоположность ПЛ/1 и АЛГОЛу W ФОРТРАН для представления указателей требует массива, параллельного массиву (или массивам), представляющему сами данные (Рис. V.25).

Здесь надо знать, где взять место, выделяемое новому элементу; эта работа выполняется фактически программистом, а не системой. Так, на Рис. V.25 «пустые» клетки могут в действительности содержать старые данные, ставшие недопустимыми (см., например, Рис. V.26).

Допустим, что необходимо добавить элемент D между A и B с тем, чтобы сформировать линейный список (A, D, B, C). Как программа может узнать, используются ли позиции 10, 9, 7, 6, 5, 3 и 1 в массиве? Можно предусмотреть несколько решений:

а) использовать элементы массива в порядке возрастания индексов, не занимаясь восстановлением свободного места до тех пор, пока массив не будет исчерпан; затем обратиться к подпрограмме **«сборщик мусора»** (см. разд. V.1.4.2);

б) использовать упрощенный метод **счетчика ссылок:** когда элемент исключается из линейного списка, его помечают как уничтоженный (например, присваивая специальное значение –1 соответствующему указателю); в поисках места для добавляемого элемента просматривают массив от начала до обнаружения помеченного элемента. Этот способ медленнее предыдущего: он занимает фиксированное время для доступа к списку, но при исчерпании массива и составлении перечня восстанавливаемых областей не требует достаточно сложной и медленной обработки, как при сборщике мусора.

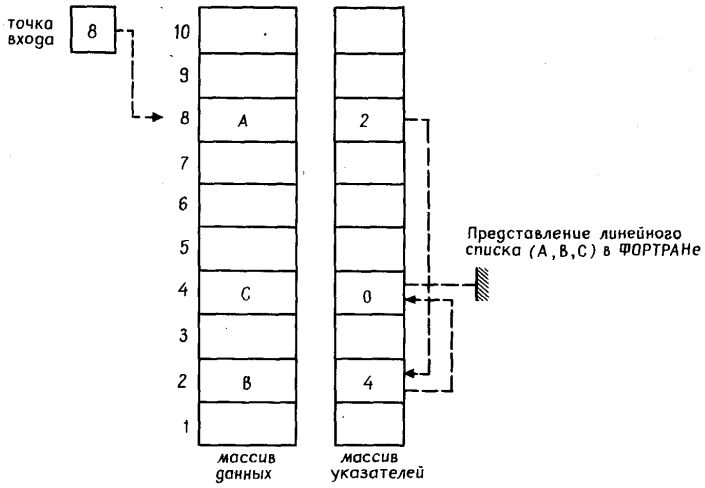


Рис. V.25 Линейный список, представленный массивами.

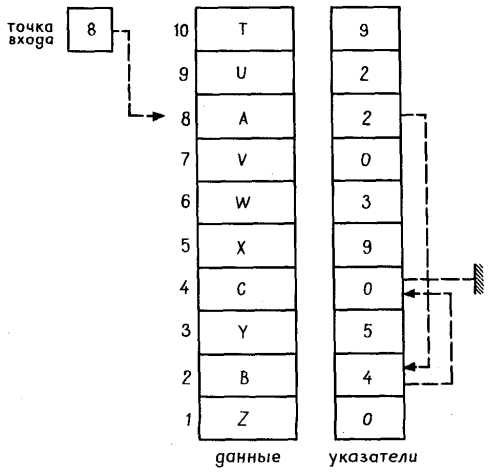


Рис. V.26 Линейный список (A, B, C) со старыми, недоступными данными.

в) рассматривать фактически управление двумя списками: списком, явно обрабатываемым в программе, и **списком свободных мест;** последний список управляется как *стек,* в котором засылка элемента в стек выполняется, когда место освобождается, а выборка – при запросе места. При этом методе нужно инициализировать стек свободных мест, соединив

между собой все элементы в произвольном порядке (Рис. V.27).

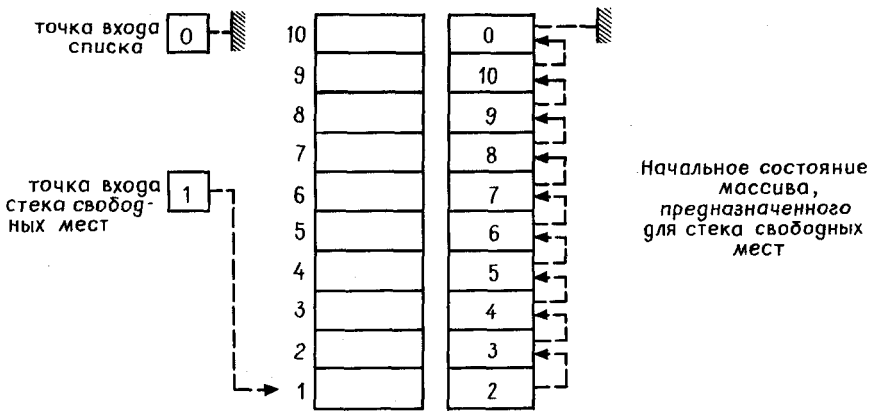
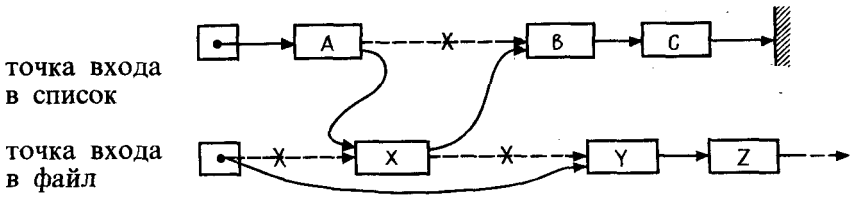


Рис. V.27 Начальное состояние массива (пример).

Значения, связанные в каждый момент с элементами стека свободных мест, несущественны; поэтому нет необходимости инициализировать массив данных.

Схема изменения указателей, приводившаяся выше, становится теперь такой:



Отметим, что элементы стека явно связаны между собой указателями, тогда как рассмотренное в разд. V.4.4.1 представление использовало связь, неявно определяемую расположением.

Линейный список (A, B, C) может соответствовать такому состоянию памяти (Рис. V.28):

В этих условиях

- чтобы прибавить элемент к списку, используют шестую позицию массива, тогда значение точки входа в стек становится равным 1;
- если удаляется, например, B из списка, то верхушкой стека становится позиция 2, связываемая с позицией 6, которая была прежней верхушкой.

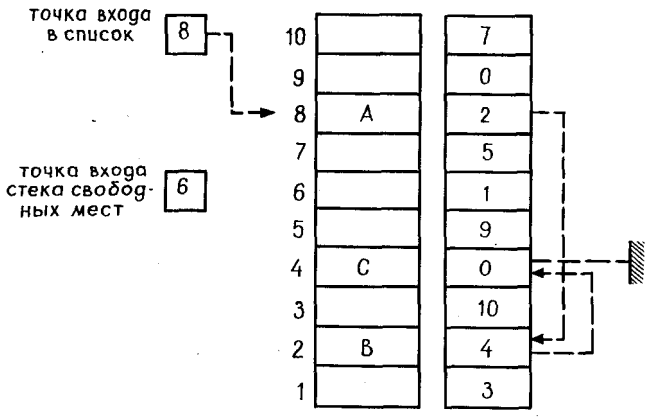
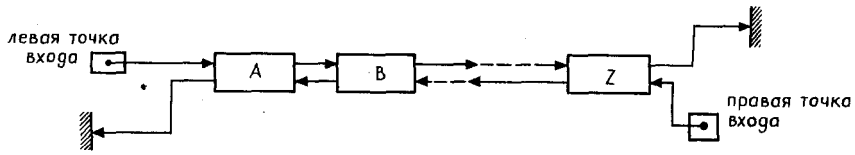


Рис. V.28 Линейный список (A, B, C) со стеком свободных мест (соединение этих мест не изображено).

Включение или удаление элемента порождает здесь изменение трех указателей: требуемое время *постоянно* и не зависит от занятости массива. Это одно из определяющих преимуществ метода. Другим преимуществом является легкость, с которой обнаруживается переполнение: переполнение имеет место тогда и только тогда, когда происходит попытка включения при точке входа в стек, равной 0.

Только что рассмотренное представление создает преимущество одному из двух

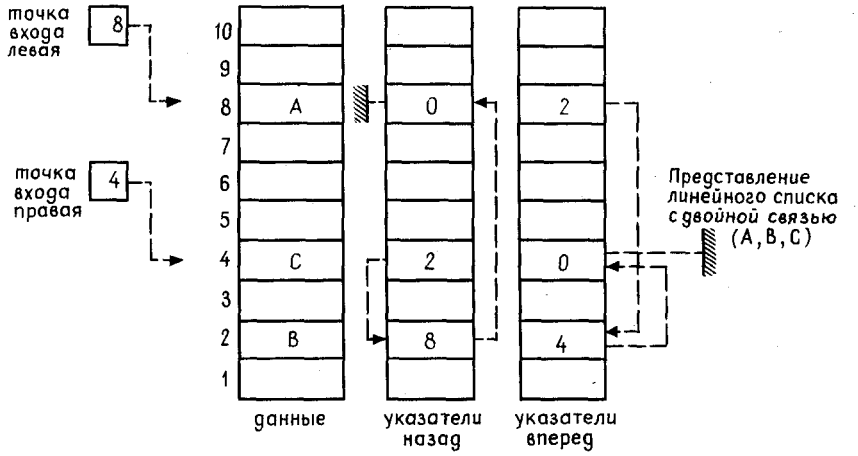
направлений прохождения стека. Со всей очевидностью оно приспособлено к «однонаправленному» использованию –сортировке включением (см. разд. VII.3.5). В других случаях, однако, сказывается отсутствие симметрии, которая может быть предусмотрена следующим представлением, называемым **двойной связью:**



Такое представление приводит к необходимости пополнения функциональной спецификации функцией « предыдущий) и позволяет проходить список в обоих направлениях с одинаковой легкостью, правда, ценой двух указателей на элемент вместо одного. Так, в АЛГОЛе W цепное представление объявляют:

```
RECORD LISTE_A_DOUBLE_LIEN (REFERENCE(T) DONNEE;
      REFERENCE(LISTE_A_DOUBLE_LIEN) GAUCHE, DROIT);
COMMENT : LISTE_ A_ DOUBLE_ LIEN – СПИСОК С_ ДВОЙНОЙ СВЯЗЬЮ,
            DONNEE – ДАННОЕ, GAUCHE – СЛЕВА, DROIT – СПРАВА;
```

а при цепном представлении, управляемом программистом, приходят к схеме:



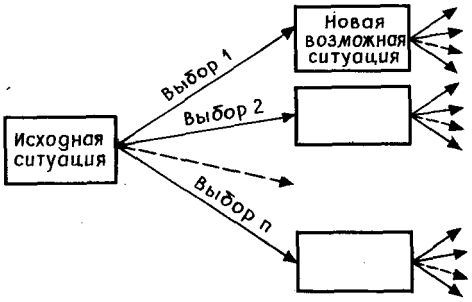
Обобщение линейных списков составляют структуры, называемые «списками»; это структуры, тесно связанные с деревьями. Они будут рассмотрены в V.8.

V.7. Деревья, двоичные деревья

V.7.1. Деревья: определение и применения

Среди структур данных наиболее важными являются **деревья;** в частности, они наилучшим образом приспособлены для решения задач искусственного интеллекта и синтаксического анализа одновременно.

В задачах *искусственного интеллекта* редко удается точно предусмотреть ход вычислительного процесса или обработки данных, идет ли речь о программе, играющей в шашки или шахматы, определяющей план действия робота, доказывающей теоремы или правильность программ или анализирующей зрительные или звуковые образы. В этом типе алгоритмов (который будет подробнее изучен в разд. VI.5) почти всегда встречаются *последовательные испытания:* на каждом этапе возможны несколько действий; каждое из них приводит к новому состоянию, откуда могут выходить еще несколько «путей». Поиск оптимальной стратегии требует, таким образом, ветвящегося процесса и построения дерева:



В *синтаксическом анализе* определяют *формальные грамматики,* которые представляют собой множество *правил подстановки.* Так, выражение можно определить как множество *термов,* разделяемых знаками + или –; в свою очередь терм есть совокупность *множителей,* разделяемых знаками * или /; наконец, множитель может быть либо *идентификатором,* либо *константой.* Иерархическая структура выражения может быть описана древовидной схемой (Рис. V.29).

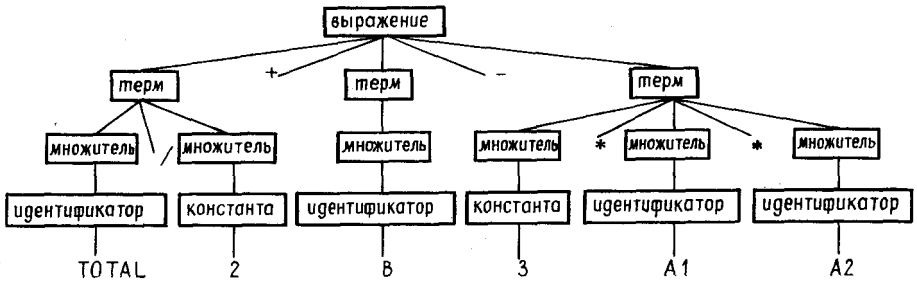
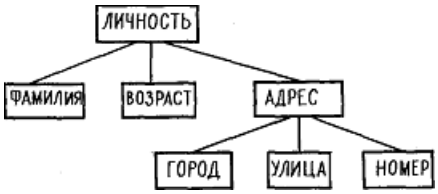


Рис. V.29 Дерево выражения TOTAL/2 + B – 3 * A1 * A2

При обработке текста программы транслятор можетиспользовать деревья для декодирования таких выражении, а также для распознавания, например, иерархических структур ПЛ/1. В разд. V.2.2 мы видели, например, структуру, соответствующую дереву:



Заметьте, что привычные соглашения заставляют деревья в информатике расти вниз головой.

Самое простое определение дерева рекурсивно:

Деревом типа T называется структура, которая образована данным типа T, называемым **корнем** дерева, и конечным, возможно, пустым множеством с переменным числом элементов–деревьев типа T, называемых **поддеревьями** этого дерева

Факт наличия упорядоченности на множестве поддеревьев данного дерева определяется в зависимости от конкретных приложений.

Так, синтаксическое дерево Рис. V.29 образовано корнем «выражение» и пятью поддеревьями.

Терминология, используемая в связи с деревьями, включает следующие понятия:

- **лист** – это корень поддерева, не имеющего, в свою очередь, поддеревьев; таковы TOTAL, +, A1, 3, A2, B на синтаксическом дереве Рис. V.29;
- **вершина** – это корень поддерева. Корень и листья являются особыми вершинами; не совпадающие с листьями вершины называются

внутренними вершинами (например, ЛИЧНОСТЬ и АДРЕС в приведенном выше примере);

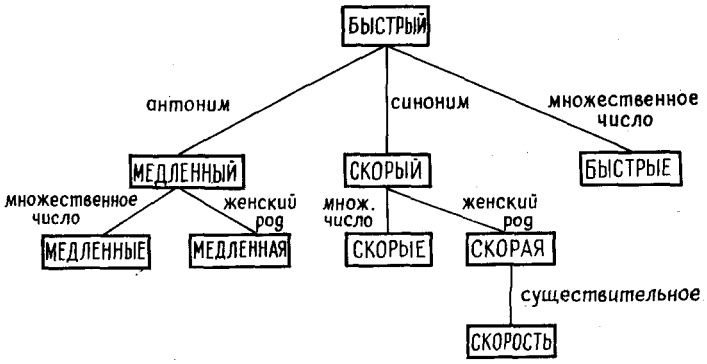


Рис. V.30 Помеченное дерево.

- поддерево называют **сыном** по отношению к своему дереву, и, наоборот, дерево считается **отцовским,** или **родительским,** по отношению к своим поддеревьям; два поддерева одного дерева становятся тогда **братьями;**
- вершина связана с каждым из своих поддеревьев **ветвью.**

Описываемая древовидной структурой информация может связываться не только с вершинами дерева, но и с его ветвями. Данные, соответствующие ветвям, называют **метками,** а дерево в таком случае считается **помеченным** (Рис. V.30).

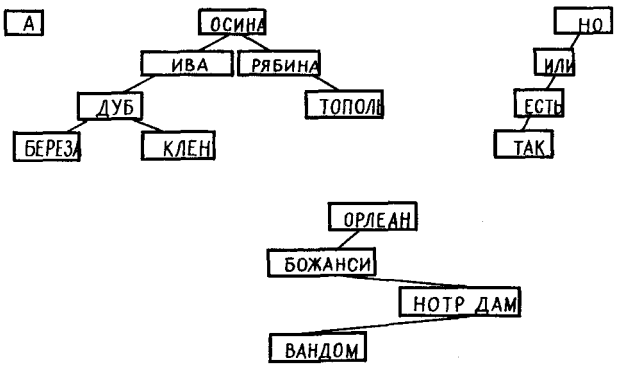
Мы не касаемся сейчас спецификации и представления деревьев, которые легче исследовать, исходя из близкой и более простой структуры–двоичного дерева, к которой, как будет показано, общие деревья могут быть сведены.

V.7.2. Введение в двоичные деревья

Двоичным деревом типа T называют структуру, которая либо **пуста,** либо образована:

- данным типа T, называемым **корнем** двоичного дерева;
- *двоичным деревом* типа T, называемым **левым поддеревом** (ЛПД) двоичного дерева;
- *двоичным деревом* типа T, называемым **правым поддеревом** (ППД) двоичного дерева

Вот несколько примеров двоичных деревьев (типом T здесь являются строки)[1]:

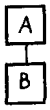


Направления линий, связывающих вершины с их поддеревьями, позволяют отличить ЛПД от ППД. Отсутствие линии указывает, что соответствующее поддерево пусто. Термины, введенные для деревьев, применимы в равной мере к двоичным деревьям.

[1] Среди этих примеров находится и пустое дерево, но оно невидимо.

На первый взгляд кажется, что двоичное дерево является не чем иным, как частным случаем дерева, каждая вершина которого всегда имеет две ветви. Главное различие состоит в том, что два возможных поддерева двоичного дерева существенно различаются. Это различие важно: пусть существуют два двоичных дерева

Первое двоичное дерево имеет ЛПД, которое является листом B, его ППД пусто; второе имеет пустое ЛПД и лист B в качестве ППД. Они, таким образом, различны, тогда как в случае деревьев они были бы идентичны и оба соответствовали бы схеме



В графическом изображении двоичного дерева «наклон» ветвей, следовательно, важен (другое различие между деревьями и двоичными деревьями состоит в том, что двоичное дерево может быть ПУСТО, тогда как дерево, рассматриваемое в общем .случае, таким быть не может).

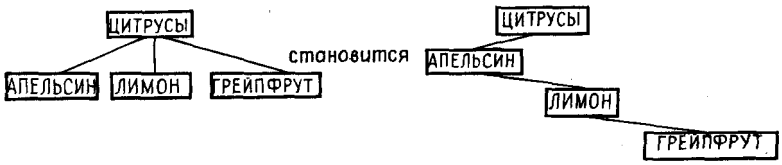
V.7.3. Преобразование дерева в двоичное дерево

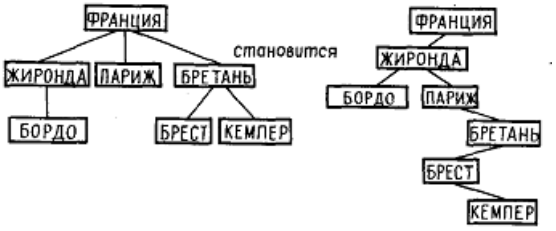
Существует «каноническое» средство каждому дереву A поставить в соответствие некоторое двоичное дерево B; оно сводится к рекурсивному применению следующих правил:

- корень B есть корень A;
- ЛПД двоичного дерева B есть *производное* от первого поддерева A, если оно существует;
- ППД двоичного дерева B есть *производное* от «младшего брата» A, если оно существует.

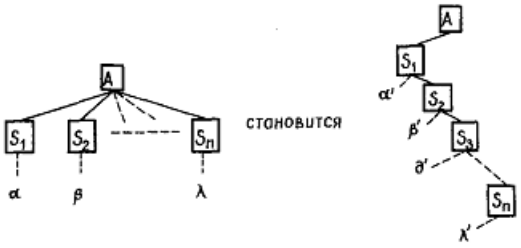
Множество поддеревьев дерева здесь предполагается упорядоченным. Третье правило не имеет смысла для дерева в целом и применяется только к поддеревьям; в результате получается, что двоичное дерево, полученное преобразованием дерева, не имеет ППД.

Примеры:





в общем виде



где α', β', ..., λ′ это двоичные деревья, полученные преобразованием деревьев α, β, ..., λ.

Предоставляем читателю убедиться, что это преобразование однозначно: для каждого дерева оно определяет соответствующее двоичное дерево; с другой стороны, двоичное дерево, корень которого не имеет ППД, есть производное некоторого дерева, и только его.

V.7.4. Функциональная спецификация

Тип $ДД_T$, или двоичное дерево типа T, определяется операциями:

созданиедерева: → $ДД_T$
деревопусто: $ДД_T$ → ЛОГ
чтениекорня: $ДД_T$ → T {получение корня непустого дерева}
слева: $ДД_T$ → $ДД_T$ {доступ к левому поддереву непустого дерева}
справа: $ДД_T$ → $ДД_T$ {доступ к правому поддереву непустого дерева}
построение: T × $ДД_T$ × $ДД_T$ → $ДД_T$ {составление дерева из корня и двух заданных поддеревьев}

операции, которые проверяют (для a и a' типа $ДД_T$, t типа T):

(Б1) деревопусто (созданиедерева) = **истина**
(Б2) деревопусто (построение^, a, a')) = **ложь**
(Б3) чтение корня (построение (t, a, a')) = t
(Б4) слева(построение(t, a, a')) = a
(Б5) справа (построение(t, a, a')) = a'
(Б6) построение(чтениекорня(a), слева(a), справа(a)) = a

V.7.5. Логическое описание

Совершенно естественно используется описание, содержащее объявления с разделением вариантов и двойным рекурсивным соединением :

тип $ДВОИЧНОЕДЕРЕВО_т$ = (ПУСТО | $НЕПУСТОЕДВОИЧНОЕДЕРЕВО_т$)
тип $НЕПУСТОЕДВОИЧНОЕДЕРЕВО_т$ = (корень : T,
лпд, ппд: $ДВОИЧНЫЕДЕРЕВЬЯ_т$)

и включающее подпрограммы

программа созданиедерева: $ДВОИЧНОЕДЕРЕВО_т$
| созданиедерева ← ПУСТО
программа деревопусто: ЛОГ (**аргумент** b: ДВОИЧНОЕДЕРЕВОт)
| деревопусто ← в **есть** ПУСТО

```
программа чтениекорня : Т (аргумент b : ДВОИЧНОЕДЕРЕВО_т)
    | если деревопусто (b) то ошибка
    | иначе чтениекорня ← корень (b)
программа слева :ДВОИЧНОЕДЕРЕВО_т (аргумент b: ДВОИЧНОЕДЕРЕВО_т
    | если деревопусто (b) то ошибка
    | иначе слева ← лпд (b)
программа справа : ДВОИЧНОЕДЕРЕВО_т (аргумент b: ДВОИЧНОЕДЕРЕВО_т)
    | если деревопусто (b) то ошибка
    | иначе справа ← ппд (b)
```

Важное замечание:

Можно увидеть, что определение типа ДВОИЧНОЕДЕРЕВО$_т$ формально идентично определению *файла с двойным доступом* (V.5.5). Это наглядно показывает, что структура данных полностью определяется только тогда, когда одновременно известны и составляющие ее элементарные объекты, и базовые функции, работающие с этой структурой, т.е. функциональная спецификация, а не только логическое описание.

В связи с двоичными деревьями (и деревьями вообще) вводится важная категория алгоритмов: алгоритмы обхода дерева. Такой алгоритм–это метод, позволяющий получить *доступ к каждой вершине дерева один и только один раз.*

Для каждой вершины выполняются некоторые виды обработки (проверка, запись, суммирование и т.д.), однако способ обхода независим от этих действий и является общим для алгоритмов, которые могут выполнять различную обработку.

Когда речь идет о двоичных деревьях, простейшие алгоритмы обхода состоят в выполнении трех следующих действий в некотором порядке:

- обработка корня;
- *обход* левого поддерева;
- *обход* правого поддерева.

Обратите внимание, что эти способы определены рекурсивно (как это подчеркивает курсив). Существует шесть возможностей, обозначаемых ЛКП (Левое; Корень; Правое), КЛП, ЛПК, ПКЛ, КПЛ, ПЛК. КЛП называют Также обходом сверху, или **префиксным** обходом, или **прямым** обходом (сначала корень, затем сыновья); ЛПК – обходом **снизу,** или **постфиксным,** или **концевым** (сначала сыновья, затем корень); ЛКП – обходом **слева направо,** или **инфиксным,** или **обратным.** Упражнение V.2 позволяет сравнить эти порядки обходов на примере.

Весьма полезным применением двоичных деревьев, которое дает пример ЛКП, является понятие **двоичного дерева поиска.** Предположим, что для элементов типа Т установлено отношение порядка; тогда можно договориться о включении в поддерево элементов, меньших, чем корень, всегда слева от некоторого поддерева, а элементов, превосходящих корень, – справа.

Программу, реализующую это, можно записать в виде

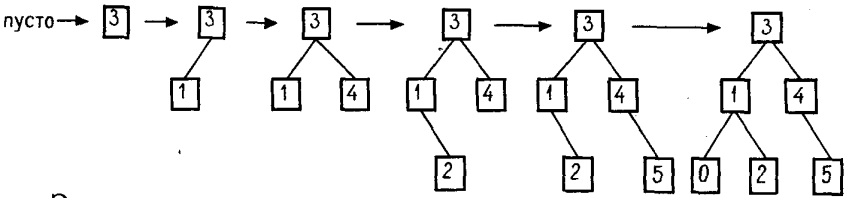
```
программа включение : ДВОИЧНОЕДЕРЕВО_т (аргументы t: Т,
                    a : ДВОИЧНОЕДЕРЕВО_т)
    | включение ← если деревопусто (a) то
    |       | построение (t, ПУСТО, ПУСТО)
    | иначе если 1 < чтениекорня(a) то
    |       | построение (корень (a),
    |       |     включение (t, лпд(a)), ппд(a))
    | иначе
    |       | построение (корень (a), лпд(a),
    |       |     включение (t, ппд(a)))
```

Элемент t «спускается» по дереву, начиная от корня, поворачивая влево и вправо

у каждой вершины в зависимости от соотношения t со значением в этой вершине, до тех пор, пока не обнаруживается пустое место. Тогда t занимает это место. Так, программа

```
переменная a : ДВОИЧНОЕДЕРЕВО_цел
a ← созданиедерева;
a ← включение(3, a); a ← включение(1, a);
a ← включение(4, a); a ← включение (2, a);
a ← включение(5, a); a ← включение (0, a);
```

следующим образом развертывает дерево a:



Этот метод упрощает поиск элемента; ясно видно, что если в дереве имеется n элементов, то число сравнений, необходимое для доступа к одному из них, есть максимум глубины дерева, т.е. $\log_2 n$ в лучшем случае, вместо n при размещении элементов в линейном списке. Этот способ сортировки является просто обходом ЛКП дерева:

```
программа обход (аргумент b : ДВОИЧНОЕДЕРЕВО_т)
    если ~деревопусто (b) то
            обход (слева (b));
            обработка (корень (b))
            обход (справа (b))
```

Действие обработка(t) может быть любым действием, которое необходимо применить к каждому элементу, например печать, если требуется получить упорядоченный список элементов. Читатель может убедиться, что программа «обход» обрабатывает все элементы по порядку; если при этом *рекурсивный* характер программы шокирует читателя, то стоит подождать разъяснений гл. VI. Алгоритмы «таблиц решений», использующих двоичные деревья и их точные характеристики, будут подробно рассмотрены в разд. VI.2.4.

V.7.6. Физическое представление

V.7.6.1. Цепное представление

Чтобы представить двоичное дерево (а значит, и дерево вообще) цепным способом, каждому его элементу ставят в соответствие два указателя – к ЛПД и ППД. **В АЛГОЛе W,** например, объявляется

```
RECORD ARBIN T(REFERENCEfT) RACINE;
                REFERENCE(ARBIN_T) SAG, SAD)
COMMENT:ARBIN_T – ДВОИЧ_ДЕРЕВО_T, RACINE – КОРЕНЬ,
SAG – ЛПД, SAD – ППД;
```

Описания подпрограмм созданиедерева, деревопусто, чтениекорня, слева, справа «перерисовываются» с их логического описания. Методы ПЛ/1 сходны.

В ФОРТРАНе с массивом элементов связывают два параллельных массива *ЛПД* и *ППД,* задающих левые и правые связи (Рис. V.31). Нулевая связь соответствует ПУСТО. Двоичное дерево Рис. V.31 есть двоичное дерево поиска, если выбранный порядок – алфавитный.

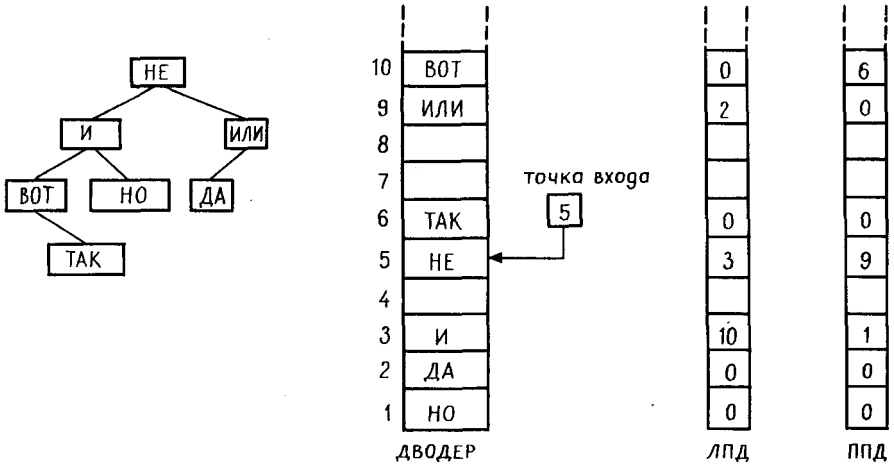


Рис. V.31 Представление двоичного дерева.

В качестве примера рассмотрим в АЛГОЛе W проверку (рекурсивную) на равенство содержимого двух двоичных деревьев. Записывают:

АЛГОЛ W

```
COMMENT : EGALITE – РАВЕНСТВО
LOGICAL PROCEDURE EGALITE (REFERENCE (ARBIN_T)
                           VALUE Bl, B2);
    (B1 = B2) OR (
        B1 ¬= NULL) AND (B2 ¬= NULL)
        AND (RACINE (B1) = RACINE (B2))
        AND EGALITE (SAG (B1), SAG (B2))
        AND EGALITE (SAD (B1), SAD (B2))
            )
```

(См. упражнение V.3.)

Заметьте, что проверка *B1 = B2* дает результат **истина,** если оба дерева пусты: вторая часть *OR* в таком случае не выполняется.

В случае двоичного дерева поиска (т.е. если используется отношение порядка на T) подпрограммы включения и поиска имеют нерекурсивные версии, более сложные, чем рассмотренные выше, но и более эффективные; мы к ним вернемся в гл. VII (VII.2.4.1 и VII.2.4.2 для «равновесных» двоичных деревьев).

V.7.6.2. Сплошное представление

Для двоичных деревьев возможно и сплошное представление. Оно особенно интересно для «полных» деревьев, т.е. деревьев, все вершины которых заняты, кроме, быть может, самых, «правых» вершин последнего уровня (Рис. V.32).

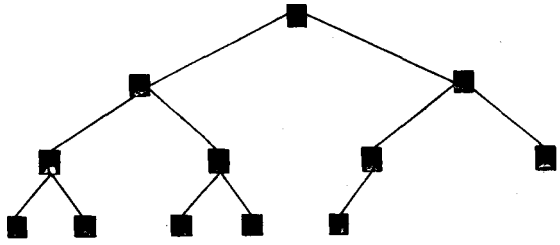
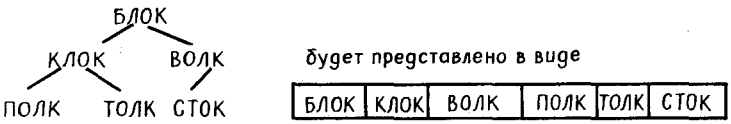


Рис. V.32 Полное двоичное дерево.

В этом случае можно использовать массив без явных связей; сыновья элемента с индексом i проиндексированы соответственно 2i и 2i + 1.

Пример:



Элемент j уровня i имеет индекс $2^{i-1} + j - 1$. Заметим, что здесь неявно использован алгоритм обхода «по уровням» слева направо, отличный от предыдущих алгоритмов (ЛКП и т.д.).

Этот метод экономичен с точки зрения места в памяти, так как для «полного» двоичного дерева с n вершинами он требует только места, необходимого для представления n объектов типа T.

V.8. Списки

V.8.1. Введение. Применения

Списки, которые не следует путать с рассмотренными выше «линейными списками», это наиболее любопытные и более других заслуживающие интереса среди всех тех существ, которые населяют Валгалу[1] информатики. Это в высшей степени рекурсивные структуры; действительно, понятия списка и рекурсии представляют столь хорошую совокупность средств, что такой язык программирования, как ЛИСП, основан почти исключительно на их комбинации.

Пусть T – множество произвольных объектов. **Список** ℓ объектов T это:

- либо элемент из множества T; в этом случае список называют **атомическим;**
- либо множество $(\ell_1, \ell_2, \ldots, \ell_n)$, где $\ell_1, \ldots, \ell_n$ *–списки.*

  Частный случай представляет собой пустой список, который обозначается ( ) или ПУСТО

Это в полном смысле рекурсивное определение непосредственно приводит к логическому описанию типа $СПИСОК_т$, обсуждение которого мы, однако, проведем немного позднее. Далее множество T, или множество атомов, будет включать слова, написанные прописными буквами и похожие на идентификаторы языков программирования, такие, как A, TOP, ATOM_1 и т.д., числовые константы, как 327, и знаки, как +, – и т.д. Примеры списков, образованных из элементов этого множества:

( ) {пустой список}
ATO (атомический список)
(ATO1 ATO2) {список из двух элементов}
((A B) C (D (E F) G))
{ список трех элементов: первый – список двух элементов, второй – атом, третий – список трех элементов, в котором второй элемент является списком двух элементов }
(( ) ( ) i)) { список трех элементов, равных пустому списку }

Важно увидеть разницу между первым примером (пустой список, список без элементов) и последним (список трех элементов, каждый из которых–пустой список). Заметьте также, что некоторые списки, как второй в приведенном примере, соответствуют линейным спискам. Третий пример подчеркивает фундаментальное свойство списков: возможность «вложенности» на произвольную глубину.

---

[1] Авторы используют малоупотребительный в отечественной литературе образ. Валгала в скандинавской мифологии это пантеон, населенный душами погибших в битвах героев. – *Прим. перев.*

Можно (и мы вернемся к этому) рассматривать списки в качестве частного случая деревьев или двоичных деревьев. Важность списков в другом: она вытекает из способности списков представлять существенно «структурированные» знания, такие, к которым должны иметь доступ программы, используемые в различных областях *искусственного интеллекта:* символические вычисления, автоматическое доказательство теорем, анализ естественных языков и автоматический перевод, программы стратегических игр, управление роботами и т.д.

Простым примером является представление **математических выражений** в символической форме. Списки позволяют описывать выражения произвольной сложности путем применения следующих правил:

- атомический список представляет переменную или константу. Примеры: X, Y, 365 и т.д.
- неатомический список представляет применение первого элемента, который должен быть знаком операции, к объектам, представляемым следующими элементами (обратите внимание на рекурсивность); например,

  (+ X Y) представляет x + y {мы называем x переменную, представляемую атомом X и т.д.}

  (× X Y Z) представляет x × y × z

  (+ (– X 3) (× Y Z)) представляет (x – 3) + (y – z)

Интересно отметить, что эту систему можно обобщить на представление не только формул, но и свойств, определенных на формулах: аксиом, теорем и т.д. Так, (=ℓ_1 ℓ_2) представляет свойство «объекты, представленные списками ℓ_1 и ℓ_2, равны»; например,

(= (+ XY)(× 3Z))

означает x + y = 3 × z

точно так же: (⇒ (= X Y) (≠ (+ Z T) U))

означает: x = y ⇒ (z + t) ≠u

(& L_1 L_2) означает «ℓ_1 и ℓ_2», где ℓ_1, и ℓ_2 – свойства, представленные L_1 и L_2. В такой системе можно писать

```
(⇒(&(ПРИНАДЛЕЖИТ X D1)
     (ПРИНАДЛЕЖИТ X D2)
     (//Dl D)
     (//D2 D))
( = Dl D2))
```

(Вы узнали аксиому Евклида?)

Любая задача программы искусственного интеллекта (здесь, к примеру, автоматического доказательства) сводится, таким образом, к обработке списков такого вида: некоторые из списков – данные, это «аксиомы»; другие должны быть получены из предыдущих, это будут «теоремы», которые программа «доказывает» с помощью дедуктивных правил (*загадка:* как представлены дедуктивные правила?).

Интересным следствием такого подхода является то, что различие между «данными» и «программой» становится менее четким: аксиома, как, например, показанная выше аксиома Евклида, может рассматриваться в двух аспектах – как информация, задаваемая программе обработки символических формул, и как подпрограмма, позволяющая в некоторых случаях доказывать равенство объектов; в этой последней интерпретации, кроме того, классическая противоположность между алгоритмом (фиксированным, или «инвариантным») и данными (переменными) постепенно исчезает, потому что новые теоремы могут доказываться путем вычислений и включаются в ранее накопленные знания.

Язык ЛИСП и производные языки (ПЛЭНЕР, ПЛАЗМА, КОНИВЕР, QA4 ...) используют эти идеи, давая одну и ту же списковую форму «программам» и «данным», допуская динамическое

создание элементов программ и т.д. Однако это, хотя и очень интересно, выходит за рамки наших намерений.

V.8.2. Функциональная спецификация

Мы заимствуем функциональную спецификацию списков из языка ЛИСП. Чтобы избежать путаницы, которая может быть вызвана скобками, ограничивающими списки, мы будем записывать параметры функций в квадратных скобках: констр[A, (B C)] будет применением определяемой ниже функции констр к атому A и списку (B C).

Прежде всего, две функции доступа:

списокпуст : СПИСОК$_т$ → ЛОГ
{ результат **истина** тогда и только тогда, когда аргумент является пустым списком }
атомический : СПИСОК$_т$ → ЛОГ
{ **истина** тогда и только тогда, когда аргумент есть атом }

Условимся, что пустой список, обозначаемый ПУСТО или ( ), является атомическим. Фактически ПУСТО рассматривается в зависимости от ситуации либо как атом, либо как неатомический список (по аналогии с квантовой механикой можно было бы говорить о «двойственной природе» ПУСТО ...).

Далее, две функции, фактически тоже являющиеся функциями доступа, но встречающиеся и в качестве функций модификации в силу *рекурсивного* характера списков:

голова : СПИСОК$_т$ → СПИСОК$_т$
{задает первый элемент списка, если список не атомический}
хвост: СПИСОК$_т$ → СПИСОК$_т$
{задает список, получаемый из исходного удалением его первого элемента, если исходный список не атомический}

например,

голова [(A B C)] = A {атом}
хвост [(A B C)] = (B C) {неатомический список}
голова [((A B) C D)] = (A B)
голова [A] и хвост [A] не определены, потому что A – атом

Будем считать, что хвост одноэлементного списка пуст.

хвост [(A)] = хвост [((A (B) C))] = ПУСТО
но голова [((A (B) C))] = (A (B) C)
поэтому голова [хвост [((A (B) C))] =((B)) C)

Заметьте, что по этому соглашению голова [ℓ] может быть списком атомическим или неатомическим, тогда как хвост[ℓ] это либо неатомический список, либо пустой.

Наконец, две функции создания:

создание списка: → СПИСОК$_Т$
{результатом функции созданиесписка является пустой список}
и констр: СПИСОК × СПИСОК → СПИСОК
{“конструирование”}

констр[x, y] – неатомический список, в котором x – первый элемент, а y – список последующих элементов:

констр [A, (B C)] = (A B C)
констр [(A B), (C)] = ((A B) C)
констр [A, ПУСТО] =(A)

По провозглашенным выше правилам второй аргумент функции констр должен

обязательно быть либо неатомическим списком, либо ПУСТО; так, констр [A,B] запрещено. Позднее мы увидим, к чему приводит снятие этого ограничения.

Последний из приведенных примеров показывает, как можно включить в список один атом или список:

A – это атом, а не констр [A, ПУСТО] = (A)

констр [(A, B), ПУСТО] – одноэлементный список ((A B))

Таким образом определены интуитивно шесть функции: списокпуст, атомический, голова, хвост, созданиесписка и констр. Более строго их свойства характеризуются аксиомами (t – произвольный элемент типа T, уподобляющийся соответствующему атомическому списку):

(СП1)	списокпуст[созданиесписка]
(СП2)	атомический[t]
(СП3)	атомический[созданиесписка]
(СП4)	~ атомический [констр[ℓ,ℓ']]
(СП5)	голова[констр [ℓ,ℓ']] = ℓ
(СП6)	хвост [консф [ℓ,ℓ']] = ℓ'
(СП7)	~ атомический [ℓ] $\Rightarrow$ констр[готова [ℓ], хвост [ℓ]] = ℓ

Упражнение:

Сравните со спецификацией стека (V.4.2).

V.8.3. Логическое описание

Ясно, что список–это либо атом, либо **соединение** некоторого числа *списков* (возможно, нулевого числа). Это удовлетворительно в качестве логического описания, но не вписывается в развертываемый до сих пор формализм, который требует, чтобы число компонентов типа, определяемого соединением, было конечным. Мы приходим, таким образом, к тому, чтобы предложить следующее логическое описание, выражающее заданную выше функциональную спецификацию, представляя список соединением («констр») его «головы» и «хвоста». Это описание приведет нас к «каноническому» представлению списков.

тип СПИСОК$_т$ = (ПУСТО|АТОМ |НЕАТОМИЧЕСКИЙСПИСОК$_т$)
тип НЕАТОМИЧЕСКИЙСПИСОКт = (начало : СПИСОК$_т$;
продолжение: СПИСОК$_т$)
тип АТОМ = T

Обратите внимание на двойную рекурсию в определении, а также на то, что в отличие от предыдущих структур НЕАТОМИЧЕСКИЙСПИСОК$_т$ полностью рекурсивен.

Непосредственно следуют базовые функции функциональной спецификации:

```
программа созданиесписка: СПИСОК_т
  | созданиесписка ← ПУСТО
программа списокпуст: ЛОГ (аргумент ℓ: СПИСОК_т)
  | списокпуст ℓ есть ПУСТО
программа атомический: ЛОГ (аргумент ℓ: СПИСОК_т)
  | атомический ← ℓ есть ПУСТО или ℓ есть АТОМ
программа голова: СПИСОК_т (аргумент ℓ: СПЙСОК_т)
  | выбрать
  |   | ℓ есть ПУСТО: ошибка,
  |   | ℓ есть АТОМ: ошибка,
  |   | ℓ есть НЕАТОМИЧЕСКИЙСПИСОК_т : голова ←
  |                                      начало(ℓ)
```

программа хвост: СПИСОК$_т$ (**аргумент** ℓ: СПИСОК$_т$)
| выбрать
| | ℓ есть ПУСТО: ошибка,
| | ℓ есть АТОМ: ошибка,
| | ℓ есть НЕАТОМИЧЕСКИЙСПИСОКт : хвост ← продолжение (ℓ)
программа констр: СПИСОКт (**аргументы** ℓ_1, ℓ_2: СПИСОК$_т$)
| констр ← НЕАТОМИЧЕСКИЙСПИСОКт (ℓ_1, ℓ_2)

Читатель заметит, что это логическое описание не соответствует в точности предыдущей функциональной спецификации, требующей, чтобы «хвост» списка не был атомическим. Чтобы ее не нарушать, надо добавить тип

тип СПИСОКИЛИПУСТО$_т$ = (НЕАТОМИЧЕСКИЙСПИСОК$_т$|ПУСТО)

и определить СПИСОК$_т$ так, чтобы «продолжение» списка было всегда ПУСТО или неатомическим списком:

тип СПИСОК$_т$ = (начало: СПИСОК$_т$; продолжение: СПИСОКИЛИПУСТО$_т$)

В этих условиях результат программы хвост и второй параметр констр будут СПИСОКИЛИПУСТО$_т$ – а не СПИСОК$_т$.

Действительно, для того чтобы придерживаться данного описания, есть и другие основания. Оно определяет тип, более общий, чем тип списков, обсуждаемых с начала этого раздела: оно позволяет «симметризовать» или «бинаризировать» списки, уничтожая всякое различие между «головой» и «хвостом» списка. Мы будем следовать терминологии ЛИСПа, называя такой симметризованный список *S–списком.* Таким образом, **S–список** это либо ПУСТО, либо атом, либо пара (соединение) двух *S–списков.* В этом последнем случае двух S–списков ℓ_1, и ℓ_2 результирующий S–список обозначается

(ℓ_1 . ℓ_2)

Примеры S–списков:

АТОМЫ
(МЕЗОН.КАТИОН)
(АНИОН. (ЯДРО.ЭЛЕКТРОН))
((А.ПУСТО). (В. (С. D)))

В каком случае S–список является просто *списком?* Для этого необходимо и достаточно, как следует из предыдущего, чтобы имел место один из трех случаев:

а) список пуст

б) список атомический

в) список неатомический, голова его – *список,* а хвост – *список* или ПУСТО, но не атом, отличный от ПУСТО (заметьте, что от двойной рекурсии уйти не удается).

Это определение может дать алгоритм:

```
программа настоящийсписок: ЛОГ (аргумент ℓ: СПИСОКт)
    {выдает значение истина тогда и только тогда, когда ℓ есть "настоящий"
    список}
    выбрать
        ℓ есть ПУСТО: настоящийсписок ← истина,
        ℓ есть АТОМ: настоящийсписок ← истина,
        ℓ есть НЕАТОМИЧЕСКИЙСПИСОКт:
            если ~настоящийсписок (начало (ℓ)) то
                настоящийсписок ← ложь
            иначе выбрать
                продолжение(ℓ) есть ПУСТО: настоящийсписок ←
                истина,
                продолжение(ℓ) есть АТОМ: настоящийсписок ←
                ложь
                продолжение(ℓ) есть НЕАТОМИЧЕСКИЙСПИСОКт:
                настоящийсписок(продолжение (ℓ))
```

Так,

АТОМ1 есть список

(АТО.ПУСТО) – список: список констр [АТО.ПУСТО] = (АТО);

(АТОМ.(ПРОТОН.(НЕЙТРОН,ПУСТО))) – список: список

констр [АТОМ, констр [ПРОТОН, констр [НЕЙТРОН, ПУСТО]]]

= констр[АТОМ, констр[ПРОТОН. (НЕЙТРОН)]]

= констр[АТОМ, (ПРОТОН НЕЙТРОН)]

= (АТОМ ПРОТОН НЕЙТРОН);

((ГЕЛИЙ .ПУСТО).(АТОМ.(ПРОТОН.НЕЙТРОН.ПУСТО))))

есть список ((ГЕЛИЙ) АТОМ ПРОТОН НЕЙТРОН);

(ГЕЛИЙ ВОДОРОД) – не список

(ГЕЛИЙ.ВОЛОРОД).(АТОМ.(МОЛЕКУЛА.ПУСТО)))

не список, потому что его первый элемент не является списком (второй элемент – список).

В более общем виде: пусть $\ell = (\ell_1\ \ell_2\ \ldots\ \ell_n)$, где $\ell_1\ \ldots\ \ell_n$ – списки (возможно, пустые или атомические). ℓ может быть представлена в виде S–списка или в **канонической форме:**

$$(\ell_1^*.(\ell_2^*.(---.(\ell_n^*.\text{ПУСТО})---)))$$

где $\ell_1^*, \ell_2^*, ---\ell_n^*$ – это *канонические формы* $\ell_1, \ell_2 --- \ell_n$ (рекурсия!).

Разумеется, атом и ПУСТО совпадают со своими каноническими формами. Так,

(ВАТЕРЛОО ВАТЕРЛОО ВАТЕРЛОО ПУСТЫННО)*

= (ВАТЕРЛОО.)ВАТЕРЛОО.(ВАТЕРЛОО.(ПУСТЫННО.ПУСТО))))[1]

(И ВЫ (БЛАГОДАТНЫЕ ЧАСЫ))*

= (И. (ВЫ.((БЛАГОДАТНЫЕ.)ЧАСЫ.ПУСТО)).ПУСТО)))[2]

Всякому списку соответствует, таким образом, единственная каноническая форма; с другой стороны, как мы видели, S–список может быть канонической формой списка, а может и не быть ею.

Каноническая форма дает точные предписания для физического представления.

---

[1] Фрагмент строки из поэмы Гюго «Возмездие».– *Прим. перев.*

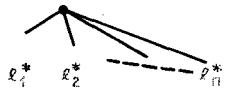
[2] Фрагмент строки из стихотворения «Озеро» в «Поэтических раздумьях» Ламартина. – *Прим. перев.*

Кроме того, она подсказывает важную аналогию между *списками* и *деревьями.*

V.8.4. Списки, деревья, двоичные деревья

Список допускает очевидное представление в виде дерева; устанавливается (рекурсивно) соответствие:

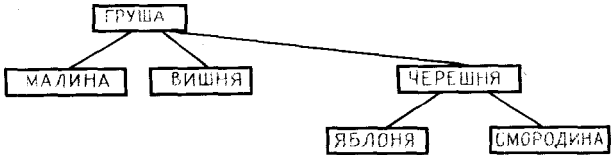
- атома A и вершины с тем же именем: A
- неатомического списка (ℓ_1, ... ℓ_n) и дерева



где $\ell_1^*, \ell_2^*, ..., \ell_n^*$ – поддеревья, соответствующие ℓ_1, ℓ_2, ... ℓ_n – ПУСТО и вершины без имени.

Так, список (МОРОШКА(КЛЮКВАОБЛЕПИХА)(КОСТЯНИКА(ЕЖЕВИКА))) представляется следующим («ягодным») деревом:

Соответственно, если задано дерево, в котором только вершины несут информацию, то с ним можно связать список, первый элемент которого есть корень, а следующие элементы соответствуют (рекурсивно) поддеревьям. Так,

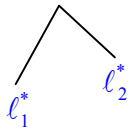


становится (ГРУША МАЛИНА ВИШНЯ(ЧЕРЕШНЯ ЯБЛОНЯ СМОРОДИНА)).

Заметьте, что это преобразование не является обратным по отношению к предыдущему, дававшему деревья, в которых только листья обладали именами; оно к тому же необратимо вообще, так как дает списки, в которых первый элемент атомический.

Со своей стороны S–списки имеют совершенно естественное представление в виде двоичных деревьев:

- ПУСТО представляется пустым двоичным деревом
- атом A соответствует вершине A
- (ℓ_1, ℓ_2) представляется

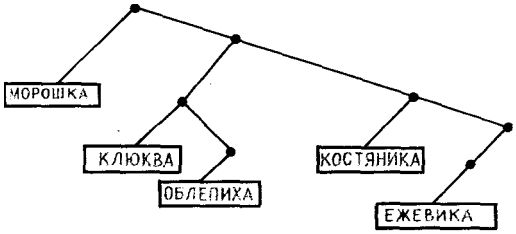


где ℓ_1^* и ℓ_2^* – двоичные деревья, соответствующие ℓ_1, и ℓ_2 (если либо ℓ_1, либо ℓ_2 есть ПУСТО, то соответствующая ветвь не будет представлена, как это обычно бывает для двоичных деревьев).

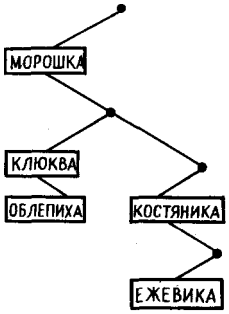
Например, S–списку в одном из предыдущих примеров:

(МОРОШКА.((КЛЮКВА,(ОБЛЕПИХА.ПУСТО)).(КОСТЯНИКА
((ЕЖЕВИКА.ПУСТО.(ПУСТО))))

соответствует двоичное дерево



Таким образом, для списка существует выбор между его представлением в виде двоичного дерева, соответствующим его канонической форме, и представлением непосредственно в виде соответствующего дерева. Читатель вспомнит (V.7.3), что всякому дереву соответствует совершенно определенным образом некоторое двоичное дерево; но (будьте внимательны!) двоичное дерево, соответствующее дереву, которое соответствует списку, не идентично двоичному дереву, соответствующему S–списку, который соответствует тому же списку! Например, наше «ягодное» дерево в качестве соответствующего двоичного дерева имеет:



Фактически двоичное дерево, соответствующее канонической форме списка, обычно ведет к его физическому представлению, которое мы сейчас будем рассматривать.

V.8.5. Физическое представление

Физическое представление списков непосредственно переписывается с логического описания. Поскольку это описание глубоко рекурсивно, представление всегда будет цепным; всякое сплошное представление превращало бы базовую операцию создания списков констр в фигуры «высшего пилотажа».

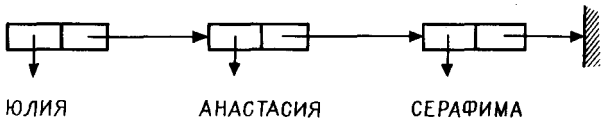
Как обычно, представление цепей в зависимости от языка определяется программистом или системой.

- Пустой список представляется специальным указателем (*NULL* в АЛГОЛе W или ПЛ/1, нулевым или отрицательным индексом в ФОРТРАНе);
- Атомический список ставит задачу представления, рассматриваемую далее;
- Базовым элементом в представлении неатомического списка является пара указателей, обозначающих голову и хвосг списка.

Так получают представление, которое есть представление двоичного дерева, соответствующего канонической форме. Например, отложив представление атомов, список

(ЮЛИЯ АНАСТАСИЯ СЕРАФИМА)

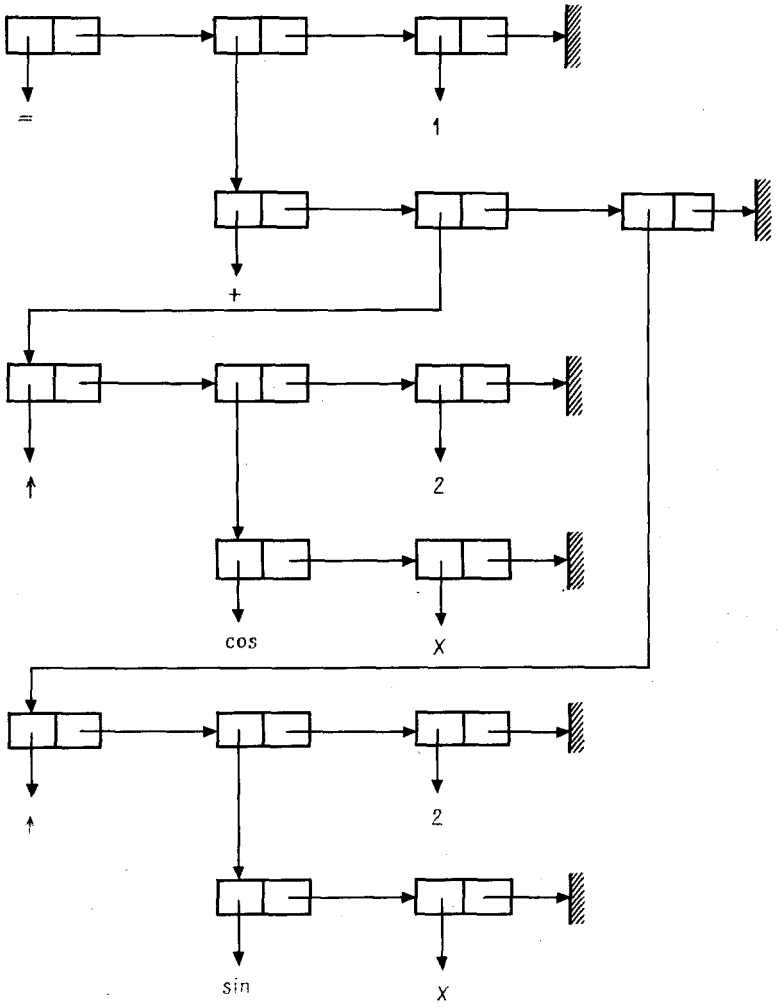
можно представить в виде



Это S–список (ЮЛИЯ.(АНАСТАСИЯ.(СЕРАФИМА.ПУСТО)))

(= (+ (↑ (COS X) 2) (↑ (SIN X) 2)) 1)

будет представлен в виде



Для атомов возникают две проблемы: нужно уметь отличать атомический список от списка, первым элементом которого является атом; кроме того, может оказаться необходимым сделать так, чтобы каждый атом представлялся только один раз. Поэтому атомический список представляется не значением его атома, а указателем, обозначающим место, которое занимает атом; нужно, кроме того, уметь проверять, является ли список атомом или нет (функция атомический в спецификации).

В АЛГОЛе W решение очень простое, поскольку представление всякой ссылки *REFERENCE* содержит указание типа обозначаемого данного.

Если, например, атомы могут быть либо целыми, либо вещественными, либо идентификаторами длиной не более 16 литер, то объявляется запись

RECORD ATOME (REFERENCE (RI, RR, RS) AT)

с объявлениями из конца разд. V.4.4.2, а список представляется записью

RECORD LISTE (REFERENCE (ATOME, LISTE) DEBUT;
REFERENCE (LISTE) SUITE)

Список *L*, объявленный ссылкой *REFERENCE (LISTE, ATOME) L*, будет атомическим тогда и только тогда, когда выражение

L = NULL OR L IS ATOME

имеет значение **истина**.

В ПЛ/1 никакой тип не соответствует указателю. Поэтому с элементом списка необходимо связывать некоторый логический индикатор (*BIT(1)*), задающий природу (список или атом) этого элемента:

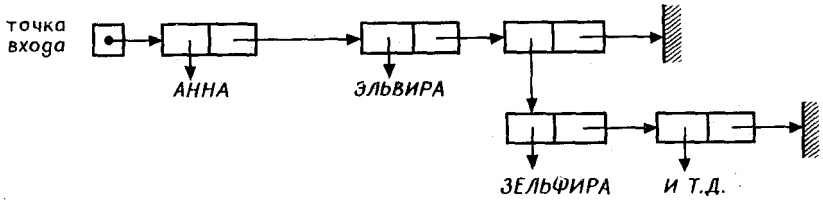
```
DECLARE 1LISTE BASED(P)
            2 DEBUT POINTER,
            2 SUITE POINTER,
            2 ATOM BIT(1)
DECLARE L POINTER,
            ATOM IQUE BIT(1)
```

В ФОРТРАНе можно точно так же каждому элементу поставить в соответствие некоторое логическое значение, чтобы указать, атомический он или нет. Простое решение состоит в представлении указателей, обозначающих атомы, их противоположными значениями: список будет неатомическим тогда и только тогда, когда он отмечается положительным указателем.

В таких условиях список может быть представлен в ФОРТРАНе двумя массивами «указателей» и одним массивом «атомов»; свободное пространство управляется при этом одним из способов, показанных для линейных списков в разд. V.6.4. Так, список

(АННА ЭЛЬВИРА (ЗЕЛЬФИРА ИТД))

который соответствует следующей схеме:



мог бы разместиться в памяти в таком виде (рис. V.33):

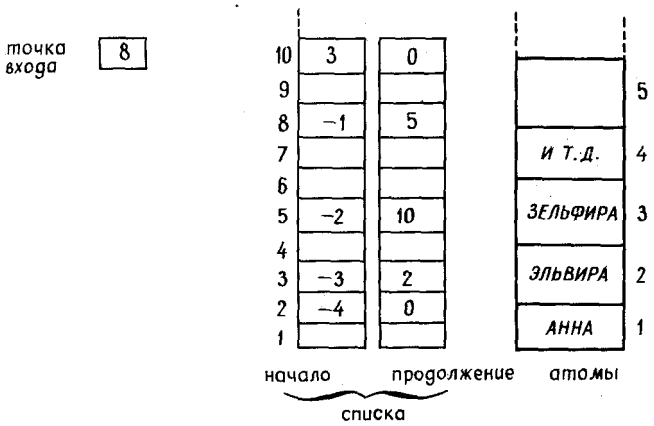


Рис. V.33 Список на ФОРТРАНе.

V.8.6. Три примера

В качестве иллюстрации применения списков рассматриваются три примера программ обработки списков. Первая программа рекурсивна: речь идет об определении **равенства** списков, т.е. установление факта, что они имеют одинаковые элементы. В АЛГОЛе W при предположении, что тип *ATOM* определен, такая программа имеет вид

АЛГОЛ W

```
RECORD LISTE (REFERENCE (ATOME,LISTE) DEBUT;
                REFERENCE (LISTE) SUITE);
LOGICAL PROCEDURE EGALITE (REFERENCE (ATOME,LISTE)
                                          VALUE L1, L2);
COMMENT: EGALITE–РАВЕНСТВО;
        ((L1 = NULL)     AND (L2 = NULL))
    OR  ((L1 IS ATOME)  AND (L2 IS ATOME) AND (L1 = L2))
    OR  ((LI IS LISTE)   AND (L2 IS LISTE)
                    AND EGALITE (DEBUT (LI), DEBUT L2))
                    AND EGALITE (SUITE (LI), SUITE (L2)))
```

Эта процедура предполагает, что атомы представлены единым способом, и, следовательно, равенство двух объектов типа *ATOM* эквивалентно равенству указателей. В противном случае надо было бы сравнивать содержимое: например, если *ATOM* объявлен как

RECORD ATOME(INTEGER CONTENU)

COMMENT: CONTENU – СОДЕРЖИМОЕ

надо заменить проверку *L1 = L2* для атомических *L1* и *L2* на

CONTENU (L1) = CONTENU (L2)

Второй пример – функция последний, которая выдает **последний элемент** (атом или список) списка:

последний [(A B C)] = C
последний [(A B) C ())] = ( )
последний [(A B (C D)) = (C D)

Рекурсивно эта функция описывается в виде

```
программа последний: СПИСОК_т (аргумент ℓ: СПИСОК_т)
| если списокпуст (ℓ) или атомический (ℓ) то
|    | последний ← ПУСТО {значение, выдаваемое но соглашению)
| иначе если списокпуст(продолжение(ℓ)) то последний ← начало(ℓ)
| иначе последний ← последний (продолжение (ℓ))
```

Не предвосхищая общие методы исключения рекурсии (гл. VI), можно дать следующий нерекурсивный вариант на ФОРТРАНе, предполагающий рассмотренное выше представление списка. Эта программа вычисляет значение индекса разыскиваемого элемента (указатель) и выдает 0, если список пуст. *ENTREE (ВХОД)* есть точка входа в список.

ФОРТРАН

```
      INTEGER FUNCTION DERN (DEBUT, SUIT E.N, ENTREE')
C     DERN – ПОСЛ, ENTRÉE – ВХОД
      INTEGER N, DEBUT (N), SUITE (N) ENTREE
C     ВЫЧИСЛЕНИЕ УКАЗАТЕЛЯ НА ПОСЛЕДНИЙ ЭЛЕМЕНТ СПИСКА 0
      INTEGER ELEM
C      —— ПЕРЕДАЧА ЗНАЧЕНИЕМ ——
      ELEM = ENTREE
      DERN = 0
C     АТОМЫ ПРЕДСТАВЛЕНЫ ОТРИЦАТЕЛЬНЫМИ ЗНАЧЕНИЯМИ, А
      ПУСТО НУЛЕМ
      IF (ENTREE. LE. 0) GOTO 1000
C                        / ПОКА ПРОДОЛЖЕНИЕ(E1:EM) |
C                        ¬= ПУСТО ПОВТОРЯТЬ /
100                      IF (SUITE (ELEM).EQ.0)GOTO 200
                             ELEM. = SUITE (ELEM)
                             GOTO 100
200                      DERN = DEBUT(ELEM)
1000    RETURN
      END
```

Наш последний пример – просто набросок. Речь идет о **символьном дифференцировании** по X математического выражения, представленного, как мы видели, в форме (+ X Y) и т.д.

Используются обычные формулы вычисления производных

$$(u\ \ v)' = u' + v';$$

$$(uv)' = u'v + uv'$$

и т.д. Предположив для упрощения, что каждая операция имеет два операнда, применяют функцию констрсписок которая конструирует список из двух или более элементов; например,

констрсписок [l_1, l_2] = констр [l_1, констр [l_2, ПУСТО]]

(не путать констрсписок с констр: констрсписок [A,B] =(A B); констр [A,B] = (A.B) – не список).

Программа имела бы следующий общий вид:

```
программа производная: СПИСОК (аргумент выр: СПИСОК)
    если списокпуст (выр) то
        производная ← ПУСТО
    иначе если атомический (выр) то
        если выр = x то
            производная ← 1
        иначе производная ← 0
    иначе если голова (выр) = + то
        производная ← констрсписок (+, производная(начало (выр)),
                        производная(начало(продолжение (выр))))
    иначе если голова (выр) = *то
        произродная ← констрсписок (+,
        констрсписок (*, производная (начало (выр)),
        начало (продолжение (выр))),
        констрсписок (*, начало(выр), производная(начало
        (продолжение (выр)))))
```

иначе ...

Дополните другие случаи.

Надо отметить, что второй элемент списка выр есть начало (продолжение(ℓ)), а не продолжение(ℓ): так, если выр = (A B C), то продолжение (ℓ) есть список (B C), а начало (продолжение(ℓ)) = B.

Заметим также, что этой программы не достаточно: ее выполнение должно сопровождаться программой упрощения, при отсутствии которой производная, полученная для (+ X 3), имела бы вид (+ 1 0), а не 1. Эта программа также предоставляется проницательности читателя.

V.9. Массивы

В гл. II было определено и широко использовалось в дальнейшем понятие массива. Цель этого раздела–дать точное описание массивов, рассматриваемых как конкретная структура данных.

Мы дадим, следовательно, функциональную спецификацию и логическое описание массивов; в равной мере мы будем говорить о возможных физических представлениях, выделяя проблемы «разреженных» массивов, т.е. массивов с малым числом значащих элементов.

Наконец, мы воспользуемся поводом, чтобы дать несколько дополнительных сведений о представлениях массивов в ФОРТРАНе, АЛГОЛе W и особенно богатым с этой точки зрения ПЛ/1.

V.9.1. Функциональная спецификация

Интуитивно мы будем характеризовать массивы как структуру с одной читающей–записывающей головкой, прямым доступом и неразрушающим чтением.

Прежде всего для известного типа T и двух целых b и B, таких, что b ≤ B, определим тип $\text{МАССИВ}_{T,b,B}$, одномерных массивов, с элементами типа T, и границами являются b и B. Пусть $I_{b,B}$ тип, определяемый перечислением как множество индексов:

тип $I_{b,B}$ = (b, b + 1, ..., B – 1, B)

Будем определять $\text{МАССИВ}_{T,b,B}$, исходя из $I_{b,B}$ и T, с помощью двух операций: доступа к элементу и модификации элемента. Удобно считать, что вторая операция модифицирует целиком весь массив, потому что в присваивании a[i] ← y нельзя знать априори, какой элемент получит новое значение. Эти операции описываются:

значение–элемента : $\text{МАССИВ}_{T,b,B} \times I_{b,B} \to T$
{для каждого индекса i из [Ь, B] определяется индекс массива }
изменение–элемента : $\text{МАССИВ}_{T,b,B} \times I_{b,B} \times T \to \text{МАССИВ}_{T,b,B}$
{всякому массиву a, индексу i и значению t ставится в соответствие массив a', i–й элемент которого равен t, а другие эле–
менты равны соответствующим элементам массива a }
создание–массива : ПУСТО $\to \text{МАССИВ}_{T,b,B}$
{создается массив типа T с границами b и B, а значения элементов
неопределенны}

Свойства (для всякого массива мас, индексов i и j из [b, B], t и t' типа T) можно сформулировать так:

(M1) значение–элемента (изменение–элемента (мас, i, t), i) = t {определение присвоенных значений}
(M2) изменение–элемента (изменение–элемента (мас, i, t), i, t') = изменение–элемента (мас, i, t')
{присваивание аннулирует предшествовавшие присваивания тому же элементу}
(M3) $i \neq j \Rightarrow$ изменение–элемента (изменение–элемента (мас, i, t), j, t') = изменение–элемента (изменение–элемента (мас, j, t'), i, t) {можно менять порядок присваивания различным элементам}

Важным моментом является, несомненно, понятие *прямого доступа:* всякий элемент полностью определяется заданием имени массива и индекса; для того чтобы до него добраться, нет необходимости в серии операций над другими элементами, как в случае стеков, списков и т.д. Именно это свойство делает массивы структурами, более «богатыми», чем предыдущие, и используемыми для описания их физического представления.

Можно ожидать, что физические представления массивов отражают это свойство, т.е. время доступа к элементу не зависит от его индекса. Как мы увидим, это не всегда так.

Ясно, что массивы могут служить для представления конечных множеств с максимальным числом $B - b + 1$ элементов. Легко реализуется операция включения; то же можно сказать о проверке наличие, если известен индекс позиции, в которой может находиться элемент, в противном случае требуется цикл. С другой стороны, массивы плохо приспособлены для задач, в которых необходима операция удаление.

Многомерные массивы можно определить рекурсивно; так, тип $МАССИВ_{T,b,b,B}$ в массивов с элементами типа T и границами $[b_1, B_1]$ $[b_2, B_2]$ может определяться как тип одномерных массивов элементов, которые сами являются одномерными массивами:

тип $МАССИВ_{T,b1,Bl,\ b2,\ B2} = МАССИВ_{МАССИВ^{T,b1,B1,b2,B2}}$

и т.д. Можно также непосредственно определить тип многомерных массивов с заданными границами

$$МАССИВ_{T,b1,Bl,\ b2,\ B2,\ldots,bn,Bn}$$

с помощью функций значение–элемента и изменение–элемента, а также аксиом M1, M2, M3, заменяя $I_{b,B}$ на $I_{b1,B1,\ldots,bn,Bn}$ множество n –ок $[i_1, i_2, \ldots, i_n]$, такое, что $b_k \leq i_k \leq B_k$ для $1 \leq k \leq n$.

V.9.2. Логическое описание

На логическом уровне массивы можно определить рекурсивно по .числу измерений:

1) Одномерным массивом типа T р границами (b, B) называется соединение $B - b + 1$ элементарных данных типа T.
2) n–мерным массивом (или массивом n измерений) типа T с границами $(b_1\ B_1)$, (b_2, B_2), ..., (b_n, B_n) $(n \geq 2)$ называется соединение $B_1 - b_1 + 1$ массивов типа T размерности $(n - 1)$ с границами (b_2, B_2), (b_3, B_3), ..., (b_n, B_n)

V.9.3. Физическое представление

Для одномерного массива с границами b и B наиболее естественное представление состоит в выделении массиву сплошной области памяти. Если предположить, что каждый элемент занимает p слов, то адресом элемента с индексом i

будет

$$a + (i - b)p$$

где a есть адрес первого элемента.

Для массива более чем одного измерения этот метод можно обобщить. Размещение называется «построчным» или «поколонным» в зависимости от того, меняются ли более быстро первые или последние индексы. Мы видели, что в ФОРТРАНе используется размещение по столбцам, в ПЛ/1 массив размещается по строкам; в некоторых других языках, в частности в АЛГОЛе W, способ размещения не уточняется стандартом языка.

Если предполагать построчное размещение, то адрес элемента $A[i_1\ i_2, ..., i_n]$ будет адресом первого элемента $A[b_1\ b_2, ..., b_n]$, увеличенным на

$$[(...(((i_1 - b_1)(B_2 - b_2 + 1) + i_2 - b_2)(B_3 - b_3 + 1) + i_3 - b_3)...)(B_n - b_n + 1) + i_n - b_n]p$$

Этот наиболее часто используемый метод не является, однако, единственно возможным. Другой подход состоит в буквальном восприятии приведенного выше рекурсивного логического описания; массив n измерений рассматривается как одномерный массив, каждый элемент которого есть массив n – 1 измерений и т.д. Например, двумерный массив задается одномерным **дескриптором** (display) (Рис. V.34).

Этот метод опускает более гибкое управление памятью, исключая необходимость резервировать большие смежные области в памяти.

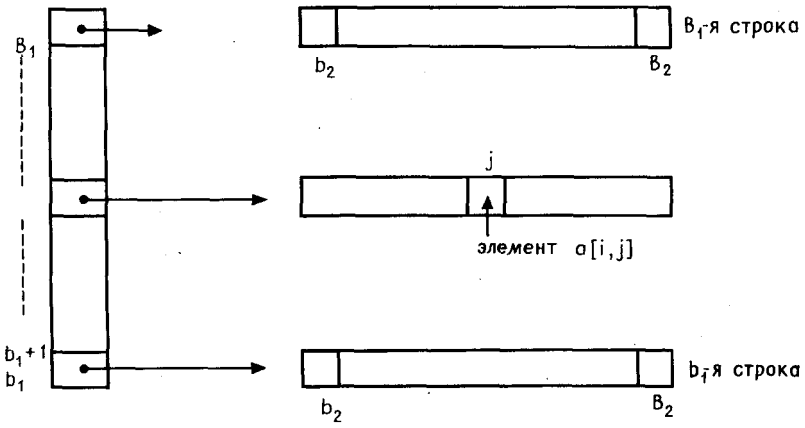


Рис. V.34 Представление $a[b_1 : B_1, b_2 : B_2]$

Но он сказывается на эффективности программ, так как требует двойного доступа к памяти при каждом обращении к элементу массива; его обобщение на n измерений требует n доступов на элемент.

V.9.4. Представление разреженных массивов

Интересен случай массивов, у которых *теоретический размер* очень велик и превосходит свободное место оперативной памяти, но большая часть элементов равна одному и тому же значению v, например (но не обязательно) нулю. Тогда ценой некоторой потери эффективности доступа можно экономить на памяти, размещая в ней только «значащие» элементы.

Способы представления таких массивов начали появляться очень рано: первые применения информатики были численными и связанными, в частности, с матричными вычислениями, при этом размеры памяти первых машин были существенно меньше по сравнению с современными ЭВМ, хранящими зачастую многие миллионы символов. Однако и сейчас проблема остается актуальной, потому что по «закону», проверенному

информатикой, потребности пользователей растут вместе с увеличением ресурсов, опережая это увеличение.

Итак, пусть по тем или иным причинам, которые нам кажутся вполне обоснованными[1], решено представить в оперативной памяти массив M, который для определенности будем полагать двумерным, с границами [1 : m, 1 : n], такими, что число

t = m · n · (размер элемента)

достаточно велико, чтобы ставить задачи, связанные с выполнением программы. Предположим, что к тому же «почти все» элементы массива M имеют одинаковое значение V; значения остальных элементов произвольные. Попытаемся использовать эту информацию для представления матрицы M в ограниченном пространстве.

Такой подход типичен (ср. разд. VII.1.2.4): когда критической характеристикой становится *пространство,* для решения задачи соглашаются на потерю *времени* (в других задачах могло бы быть принято обратное положение). Конкретно здесь это означает, что приходится частично жертвовать «прямым доступом»–этим самым удобным свойством обычного представления массивов–и, возможно, платить за это ценой цепного доступа, вызванного представлениями менее «богатых» структур. Решение является, таким образом, компромиссом между привычным представлением массивов и представлением линейных списков.

Каждый элемент матрицы M[i, j] может характеризоваться триплетом [i, j, значение M[i, j]]. Идея, очевидно, состоит в том, чтобы рассматривать только триплеты, соответствующие отличным от v элементам M, и размещать их в более простой, чем массив, структуре данных. Тогда решаемая задача сводится к реализации обычных операций для этой структуры.

Самое простое решение состоит в использовании линейного списка триплетов; элементы могут размещаться в нем в произвольном порядке или, более вероятно, упорядоченными по некоторому критерию: по возрастанию сначала первого, а потом второго индекса или, наоборот, по возрастанию M[i, j] и т.д. Кроме того, следует предусмотреть «дескриптор», содержащий значение по умолчанию v и границы m и n

Пример: пусть имеется (небольшая) матрица:

$$M = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 \\ 1 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 9 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}$$

Ее можно представить с помощью

$$\begin{cases} m = 10 \\ n = 10 \\ v = 10 \end{cases}$$

[1] Это не просто стилистический оборот: сколько трудностей представления сложных структур исчезло, когда к анализу задачи удавалось подойти под слегка измененным углом зрения.

и линейного списка триплетов

([1, 9, 6], [2, 1, 1], [2, 5, 4], [4, 7, 5], [6, 7, 3],
[8, 3, 2], [8, 4, 9], [9, 7, 7], [10, 8, 8])

Чтобы прочитать или изменить элемент массива, надо предварительно найти его последовательным просмотром списка; если в этом списке нет триплета, начинающегося с [i, j]. то значение элемента равно v.

В качестве примера использования такого представления рассмотрим сложение двух массивов. Классический алгоритм записывается в виде

```
программа сумма : массив [1 : m, 1 : n]
    (аргументы массивы M1 M2 [1 : m, 1 : n])
  | для i от 1 до m повторять
  |     | для j от 1 до n повторять
  |     |     | сумма [i, j] ← M1 [i, j] + M2 [i, j]
```

Этот алгоритм катастрофически неэффективен для больших массивов в цепном представлении, так как требует $m \cdot n$ просмотров списков M_1 и M_2. Поэтому предпочтительно применить свойства этого представления, использовав «управление» с помощью данных. Предположим, что триплеты упорядочены по индексам, второй индекс возрастает быстрее, t – разреженный массив, v(t) – значение его элементов по умолчанию.

```
программа сумма: разреженный массив [1 : m, 1 : n]
  | (аргументы M1 M2 : разреженные массивы [1 : m, 1 : n])
  | {массивы представлены списками триплетов}
  | v(сумма) ← v(M1) + v(M2) {значение по умолчанию};
  | сумма ← ПУСТО; {пустой список}
  | выбрать и повторять
  |     | {из этого списка выходят, когда исчерпаны списки M1 и M2}
  |     | ~списокпуст(M1) и списокпуст(M2):
  |     |     | взять триплет [i, j, v1] из M1;
  |     |     | прибавить к сумме триплет [i, j, v1 +v(M2)],
  |     | списокпуст(M1) и ~списокпуст(M2):
  |     |     | взять триплет [i, j, v2] из M2;
  |     |     | прибавить к сумме триплет,
  |     | ~списокпуст(M1) и ~ списокпуст(M2):
  |     |     | пусть [i1, j1 v1] – первый триплет M1;
  |     |     | пусть [i2, j2, v2] – первый триплет M2;
  |     |     | выбрать
  |     |     |     | i1 = i2 и j1 = j2:
  |     |     |     |     | прибавить к сумме триплет
  |     |     |     |     | [i1, j2, v1 + v2]
  |     |     |     |     | взять из Mj его первый триплет;
  |     |     |     |     | взять из M2 его первый триплет,
  |     |     |     | i1 < i2 или (i1 = i2 и j1 < j2):
  |     |     |     |     | прибавить к сумме триплет
  |     |     |     |     | [i1, j1, v1 + v(M2)];
  |     |     |     |     | взять из M1 его первый триплет,
  |     |     |     | i2 < i1, или (i2 = i1, и j2 < j1):
  |     |     |     |     | прибавить к сумме триплет
  |     |     |     |     | [i2, j2, v(M1) + v2];
  |     |     |     |     | взять из M2 его первый триплет
```

Использование конструкций **выбрать** и **выбрать и повторять** (ср. III.3.2) не является необходимым, но здесь оно позволяет более изящную обработку – более

симметричную, чем комбинация **пока** и **если ... то ... иначе**.

> Здесь в равной степени было бы возможно решение с помощью *сопрограмм* (IV.7); заметьте, в частности, сходство с примером, рассмотренным в IV.7.2 (слияние с суммированием двух упорядоченных последовательных файлов).

Это представление можно улучшить, использовав «полуцепное» представление, при котором первый элемент списка может быть найден более или менее прямым доступом. Это иллюстрирует Рис. V.35.

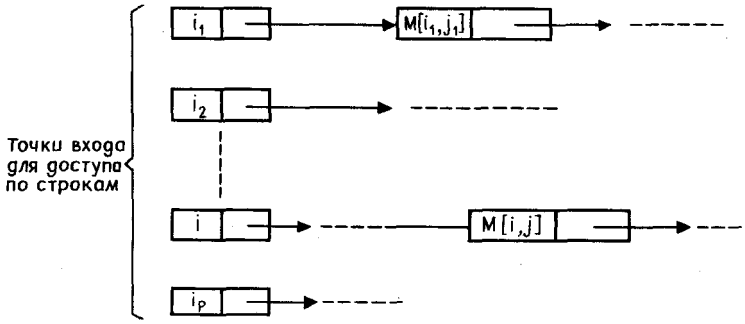


Рис. V.35 Цепное представление с доступом по строкам.

Первые элементы каждой строки могут быть объединены в цепь; в этом случае обращение к некоторому элементу порождает физическое обращение к m + n элементам, по крайней мере (поиск первого элемента строки + поиск элемента в строке). Можно попытаться устроить для них прямой доступ:

массив [1 : m] точки входа : ЛИНЕЙНЫЙСПИСОК$_т$

Это обеспечивает поиск по крайней мере за n обращений ценой возможной потери места при большом числе пустых строк. Здесь следует тщательно взвесить соотношение место–время.

Можно предложить симметричное представление, где доступ осуществляется по столбцам. Неудобство состоит в том, что в разных случаях может оказаться необходимым то или другое из этих представлений. Например, для вычисления матричного произведения

```
для i от 1 до m повторять
    для j от 1 до p повторять
        произведение [i, j] ← 0;
        для k от 1 до n повторять
            произведение[i, j] ← произведение [i, j] + M1[i, k] × M2[k, j]
```

одна из матриц обрабатывается по строкам, другая–по столбцам. Можно было бы, конечно, специализировать их представления для возможных применений, но с методологической точки зрения это неразумно и может вызывать дорогостоящие сортировки, если массив используется последовательно в разных целях. Одно возможное представление, которое делает еще шаг по направлению к прямому доступу, обеспечивает доступ сразу и по строкам, и по столбцам, как показано на Рис. V.36.

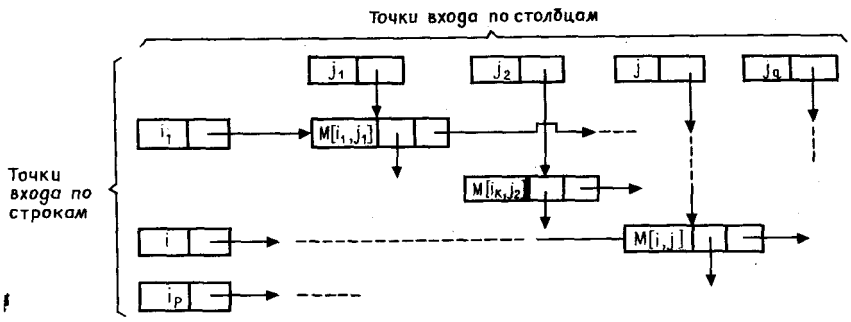


Рис. V.36 Доступ по строкам и по столбцам.

Заметим, что все эти примеры пробивают некоторую брешь в принципе доступа к структурам,

основывающимся исключительно на операциях, составляющих часть функциональной спецификации, и на независимости алгоритмов обработки по отношению к физическому представлению этих операций: алгоритмы сложения и перемножения матриц, эскизно описанные выше, совершенно иные при представлении разреженных массивов.

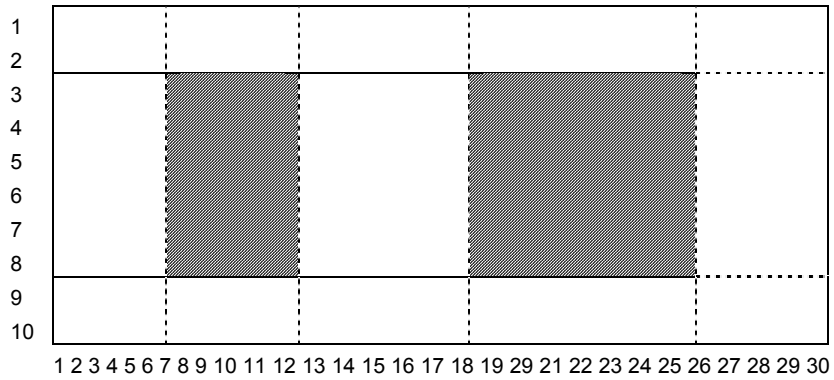
Назовем, наконец, решение, в корне отличающееся от предыдущих, – **ассоциативную адресацию.** Идея его такова: «виртуальное» пространство, определяемое массивом, имеет размер m × n элементов. «Реальное» используемое пространство существенно меньше; предположим, что оно имеет размер p, p ≥ k, где k – максимальное число значащих элементов. Обычное обозначение элемента использует неявно прямой доступ в виртуальном пространстве: a[i, j]. Это сводится к прямому доступу в реальном пространстве путем преобразования

x ← f(i, j)

где f – *функция расстановки,* такая, что 1 ≤ f(i, j) ≤ p для 1 ≤ i ≤ m и 1 ≤ j ≤ n. Задача состоит, очевидно, в нахождении функции f, обеспечивающей хорошее рассеивание результирующих адресов и способ разрешения *коллизий,* т.е. ситуаций, в которых f(i, j) = f(i',j') при различных парах индексов [i, j], [i', j']. Простейший метод состоит в объединении в цепь всех элементов, соответствующих одному значению f(i, j); этот способ действительно является шагом вперед по отношению к представлению в виде простого линейного списка, только если число коллизий остается небольшим. Эти методы ассоциативной адресации будут изучены в более общем контексте в разд. VII.3.

V.9.5. Массивы и языки: дополнения

Три наших языка предлагают, как мы видели, некоторый способ «групповой» обработки массива: это передача подпрограмме массива целиком или при некоторых условиях подмассива. Мы интересуемся при этом операциями, относящимися ко всем элементам массива в одной и той же программе. Их всегда можно промоделировать, реализуя повторяющиеся обращения к элементам благодаря *циклам со счетчиком* (см. разд. III.1). Так, чтобы прибавить 1 к заштрихованным элементам изображенной ниже фигуры, относительно которой предполагается, что она представляет вещественный массив TABLE:



записывают

ФОРТРАН

```
                REAL TABLE (10, 30)
                  .
                  .
                  .
                DO 3 I = 3, 8
                  DO 1 J = 7, 12
                    TABLE(I, J) = TABLE(I, J) + 1.
 1                  CONTINUE
                  DO 2 J = 19, 26
                    TABLE(I, J) = TABLE(I, J) + 1.
 2                  CONTINUE
 3                CONTINUE
                  .
                  .
                  .
```

АЛГОЛ W

```
REAL ARRAY TABLE (1::10, 1::30)
  .
  .
  .
FOR I = 3 UNTIL 8 DO
    BEGIN
      FOR J := 7 UNTIL 12 DO TABLE(I, J) := TABLE(I, J) + 1.;
      FOR J := 19 UNTIL 26 DO TABLE(I, J) := TABLE(I, J) + 1.
    END
  .
  .
  .
```

и в ПЛ/1 (где синтаксис повторений более свободен)

ПЛ/1

```
DCL TABLE(10, 30) BIN FLOAT
    .
    .
    .
DO I = 3 TO 8;
      DO J = 7 TO 12, 19 TO 26,
            TABLE(I, J) = TABLE(I, J) + 1.;
            END
      END;
    .
    .
    .
```

Однако ПЛ/1 предлагает дополнительные возможности, которые позволяют программисту не писать некоторые циклы, относящиеся ко всему массиву в целом; так, сумма элементов массива, которая вычисляется в ФОРТРАНе и в АЛГОЛе W посред-

ством циклов, в ПЛ/1 будет просто обозначена с помощью встроенной функции *SUM* («сумма»):

ПЛ/1

```
DECLARE TOTAL BINARY FLOAT,
TABLE (WOO) BINARY FLOAT;
    .
    .
    .
TOTAL = SUM (TABLE);
    .
    .
    .
```

Таким же образом можно использовать функцию *PROD* которая дает произведение элементов массива.

В ПЛ/1 можно также исключить явное написание циклов, предназначенных для обработки части массива, которая получается, когда одна часть индексов получает постоянное значение, а другие индексы изменяются между своими граничными значениями. Так, для того чтобы переписать первый столбец массива *TABLE* в вектор *VECT,* объявленный *VECT(10)* достаточно написать в ПЛ/1

```
VECT = TABLE(*, 1);
```

и эта возможность расширяется в таких операторах, как

```
TABLE(1, *) = TABLE(2, *) + TABLE(3, *) – 1;
```

который заменяет первую строку массива на поэлементную сумму первой и второй строк, уменьшенную на 1.

Можно комбинировать эти два способа в одном выражении. Например, *SUM (TABLE (10, *))* дает сумму всех элементов второй строки массива *TABLE*.

ПЛ/1, который не беден средствами и разнообразными обозначениями (читатель сам оценит их преимущества и недостатки), предлагает еще третью возможность, чтобы передать транслятору работу по написанию некоторых циклов; этот способ позволяет «извлекать» элементы из массива в беспорядке или через один и т.д.

Речь идет об использовании **псевдоиндексов** вида *iSUB,* которые будут только проиллюстрированы на нескольких примерах. Предположим, что *A* и *B* – двумерные массивы. Размеры их равны между собой и одинаковы для *A* и *B*. Если надо присвоить массиву *B* транспонированный массив *A* (т.е. строками *B* становятся столбцы *A),* можно написать совсем просто:

```
DECLARE A(10, 10) ...,
B(10, 10) ...,
C(10, 10) ... DEFINED A(2SUB, 1SUB);
...
B = C
```

"SUB" есть сокращение от "subscript" («индекс»). Тогда третье из приведенных объявлений должно интерпретироваться следующим образом: элементы *C(i, j)* массива *C* являются элементами *A(j, i)* массива *A,* так как *j* есть второй индекс (*2SUB*), а *i* – первый (*1SUB)*. Предположим также, что нужно образовать одномерный массив D, размер которого равен размерам массива *A*, равным между собой. Массив *D* содержит элементы *диагонали A,* т.е. элементы, оба индекса которых равны. Тогда получают *D(i) = A(i, i)* для *i* от 1 до размера *D* и записывают

```
DECLARE A(10, 10) ...;
DECLARE E(10) ... DEFINED A(1SUB, 1SUB)
DECLARE D(10) ... ;
...
D = E;
```

Наконец, если желательно поместить в вектор *P* элементы *D* с *четным индексом,* то пишут

```
DECLARE F(15) ... DEFINED D(1SUB*2);
...
P = F
```

поскольку тождество *F(i) = D(2i)* будет проверяться для всех значений *i,* которые могут быть индексами в *F*.

Кроме очевидных преимуществ, вызываемых лаконичностью (и, следовательно, ясностью) записываемых операторов, использующих эти три возможности ПЛ/1, следует назвать и то обстоятельство, что рабочая программа, создаваемая транслятором по этим операторам, вообще говоря, *короче* как по числу операторов, так и по времени выполнения, чем программа, создаваемая из соответствующих явных циклов.

Важно, однако, подчеркнуть, что неосторожное использование этих способов может привести к непредвиденным результатам, причины которых трудно обнаружить.

Заметим, что язык APL предлагает гораздо более методично, чем ПЛ/1, возможность работы с массивами целиком без использования циклов. См. [Катцан 70].

V.10. Графы

V.10.1. Определения. Графические представления

Графом называют подмножество декартова произведения двух множеств, т.е. множество пар, первый элемент которых принадлежит первому множеству, а второй элемент – второму множеству («граф» – это синоним «отношения» в математическом смысле).

Пусть, например, A и B множества

A = {Вивальди, Шопен, Паганини}

B = {фортепиано, скрипка, симфонический оркестр, камерный оркестр}

Примером графа на A × B может служить

{[Вивальди, скрипка], [Вивальди, камерный оркестр], [Паганини, скрипка], [Паганини, симфонический оркестр], [Шопен, фортепиано], [Шопен, симфонический оркестр]}.

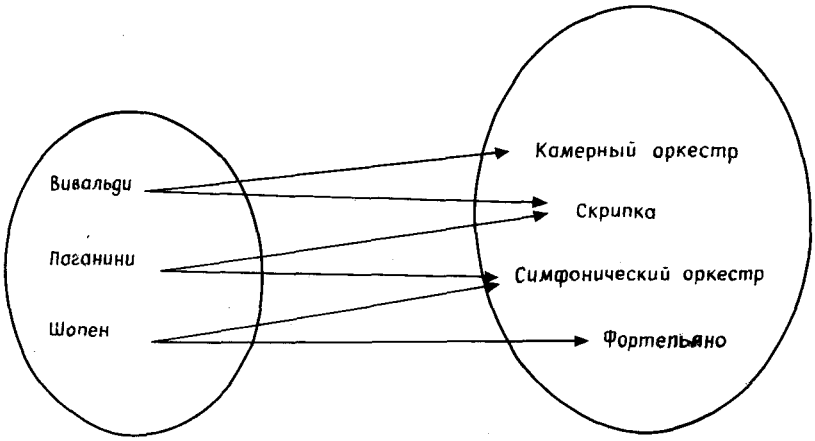


Рис. V.37 Диаграмма связей.

Этот граф конечен (он имеет шесть элементов, шесть перечисленных пар). Граф может быть и бесконечным; тогда он определяется механизмом алгоритмического построения. Например, пары целых [a, b], такие, что a – делитель b, образуют граф на $N \times N$.

Конечный граф может быть изображен двумя естественными способами. Первый из них – **диаграмма** связей, стрелки которой представляют соответствия между элементами двух множеств (Рис. V.37).

Второй способ изображения графов – **таблица соответствия**[1], отмечающая существующие соответствия среди всех возможных (Рис. V.38).

	Камерный оркестр	Скрипка	Симфонический оркестр	Фортепьяно
Вивальди	X	X		
Паганини		X	X	
Шопен			X	X

Рис. V.38 Таблица соответствия.

В случае бесконечных графов,этим способам изображения соответствуют алгоритмы, которые перечисляют элементы из B, связанные с элементом из A, или определяют принадлежность [a, b] графу.

Несколько терминов:

- элемент первого или второго множества называется **вершиной;**
- если пара [a, b] принадлежит графу, ее называют дугой $a \rightarrow b$ и изображают стрелкой в диаграмме связей.

Интересным случаем является совпадение первого и второго множеств. В этом случае **путь** от a к b есть последовательность одной или нескольких дуг, таких, что первая из них соединяет вершину a с вершиной a_1, вторая – a_1 с a_2, ..., последняя – a с b, где $a_1, a_2, ..., a_n$ – вершины графа (более строгое определение рекурсивно: путь от a к b есть либо дуга $a \rightarrow b$, либо последовательное соединение дуги $a \rightarrow c$ и *пути* от c к b, где c – произвольная вершина. **Длиной** пути считается число дуг, составляющих этот путь.

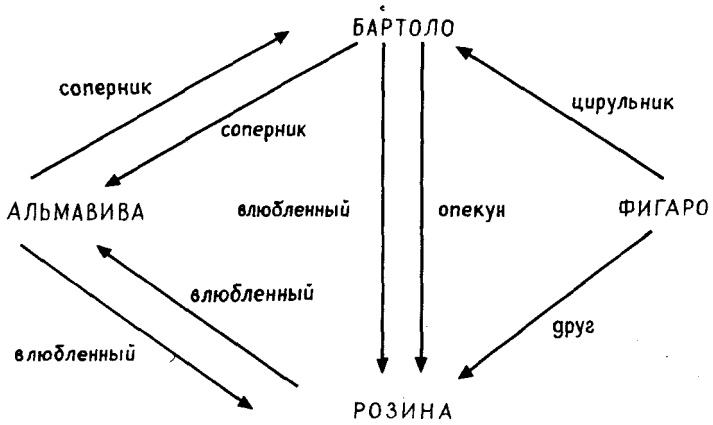
Петля есть путь, возвращающийся из некоторой вершины в нее же.

Когда первое и второе множества равны, в графе могут присутствовать пары как вида [a, b] , так и вида [b, a] . Граф называется **симметричным,** если [b, a] принадлежит графу всегда, когда ему принадлежит [a, b] .

Граф называется **помеченным,** если его дугам поставлены в соответствие некоторые данные; такой граф соответствует ситуациям, в которых недостаточно знать;

[1] В отечественной литературе по графам чаще употребляется другой эквивалентный термин – «матрица смежности». Одновременно следует отметить, что термин «диаграмма связей» нельзя считать устоявшимся. – *Прим. перев.*

что два элемента связаны некоторым отношением: важно знать еще точную природу этого отношения. Введенные выше представления легко поддаются этому расширению; например, граф отношений между людьми может дать такую диаграмму связей:



и таблицу соответствия

	БАРТОЛО	АЛЬМАВИВА	ФИГАРО	РОЗИНА
БАРТОЛО		соперник		влюбленный опекун
АЛЬМАВИВА	соперник			влюбленный
ФИГАРО	цирульник			друг
РОЗИНА		влюбленная		

Мы не даем для графов ни функциональной спецификации, ни «логического описания, которые являются строго математически определенными объектами; перейдем непосредственно к проблемам физического представления.

V.10.2. Физическое представление конечных графов

Различие между «диаграммой связей» и «таблицей соответствия» обнаруживается в двух способах физического представления конечных графов.

Таблица соответствия графа может представляться массивом в смысле языков программирования. Если два множества имеют соответственно тип элементов и если граф не помечен, то его можно тогда представить с помощью

массив граф [1 : m, 1 : m] : ЛОГ

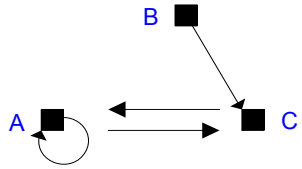
где граф [i, j] будет иметь значение истина тогда и только тогда, когда 1–й элемент первого множества связан с j–м элементом второго. Если граф помечен, то тип ЛОГ заменяется на тип T меток (если между двумя элементами могут иметь место несколько таких различных помеченных отношений, как БАРТОЛО → РОЗИНА в последнем примере, то элементами T являются линейные списки). В случае когда m = n граф является симметричным тогда и только тогда, когда симметрична соответствующая матрица.

Представление с помощью **диаграммы** связей состоит в привязке к каждому элементу линейного списка ему соответствующих элементов. Но это для нас уже не ново; речь идет просто о представлении таблицы соответствия цепным способом. Таким образом, выбор между двумя представлениями, так же как и для разреженных массивов, является вопросом соотношения пространства–времени. Если граф «разрежен», т.е. среди всех возможных отношений реально существуют лишь немногие и если m и n велики, то рекомендуется цепное представление.

Пусть непомеченный граф таков, что первое и второе множества не различаются. Его можно представить квадратной матрицей M типа ЛОГ; например,

$$M = \begin{bmatrix} \text{истина} & \text{ложь} & \text{истина} \\ \text{ложь} & \text{ложь} & \text{истина} \\ \text{истина} & \text{ложь} & \text{ложь} \end{bmatrix}$$

есть матрица графа



Два элемента i и j связаны в графе дугой тогда и только тогда, когда M[i, j] равно **истине,** i и j связаны путем длины 2 тогда и только тогда, когда существует k такое, что

M[i, k] **и** M[к, j] = **истина**

Легко видеть, что это свойство верно тогда и только тогда, когда M^2 есть **истина** M^2 – это квадратная логическая матрица, такая, что

$$M[i,j] \sum_k^n M[i,j] \cdot M[j,k]$$

где сумма представляет логическое **или**, произведение – логическое **и**, а n – размер матрицы (здесь он равен трем).

Точно так же i и j соединены путем длины 3 тогда и только тогда, когда M^3 [i, j] есть **истина,** M^3 – логический куб матрицы M. В общем случае существует путь произвольной длины между i и j тогда и только тогда, когда M^+[i, j] равно истине, где M^+ – матрица вида

$$M^+ = M + M^2 + M^3 + ...$$

Легко доказывается, что эта сумма имеет только конечное число значащих слагаемых: $M^p = M^n$ для $p > n$; действительно, если существует путь между i и j, должен существовать путь между i и j длины, не превосходящей n. Кроме того, если A – логическая матрица, то A + A = A. Можно поэтому ограничиться n членами:

$$M^+ = M + M^2 + ... + M^n$$

Матрица M^+ называется транзитивным **замыканием** M и может быть вычислена тривиальным образом с помощью матричных сумм и произведений, которые в ФОРТРАНе и АЛГОЛе W программируются циклами, а в ПЛ/1 – специальными методами, рассмотренными в V.9.5; существует, однако, значительно более эффективный алгоритм, называемый алгоритмом Уоршала, который имеет следующий вид:

```
программа замыкание (модифицируемые параметры массив R[1:n, 1:n] : ЛОГ)
    {вычисление R+ методом Уоршала}
    для i от 1 до n повторять
        {инвариант цикла: если в исходной матрице p и q были связаны путем,
        проходящим только через точки с индексами < i, то R[p, q] = истина}
        для j от 1 до n повторять
            если R[i, j] то
                для k от 1 до n повторять
                    R[j, k] ← R[j, k] или R[i, k]
```

(*Упражнение:* доказать правильность «инварианта» и алгоритма. Сравните эффективность этого алгоритма и тривиального алгоритма замыкания.)

Знание M^+ часто бывает необходимым на практике: можно искать все элементы, связанные с некоторой вершиной путем произвольной длины (что дает «класс эквивалентности» элемента, если речь идет о графе отношения эквивалентности). Часто

бывает необходимо определить, содержит ли граф петли: это возможно тогда и только тогда, когда существует i, такое, что M^+ [i, i] = **истина,** т.е. элемент истинен на диагонали M^+.

БИБЛИОГРАФИЯ

Наиболее полное описание различных структур данных содержится в гл. 2 книги [Кнут 68]. В ней, однако, не надо искать очень тонких различий между функциональными, логическими и физическими свойствами структур.

Понятие абстрактной структуры данных, не зависящей от возможных представлений, было развито Дисковом и Зилем [Дисков 74], [Дисков 75] в продолжение статей Парнаса [Парнас 72] и стало в настоящее время предметом многочисленных исследований (ср., например, [Кемен 76]). Представление, принятое в этой главе, было изложено в [Мейер 76]; подход, близкий к функциональным спецификациям, можно найти в [Гуттаг 77].

Этой проблеме была посвящена конференция в 1976 г. [ACM 76]. В ее материалах можно прочитать, в частности, статьи Кос–тера, Парнаса, Росса; другие доклады этой конференции были опубликованы не в этом сборнике, а в июньском номере Communication of ACM.

По поводу списков и их использования следует обращаться к работам по ЛИСПу, например [Маккарти 62] и [Сиклосси 76].

Две французские работы, появившиеся, когда эта книга была в печати, исследуют основные структуры данных и их приложения: [ПВР 77] и [Мал 77].

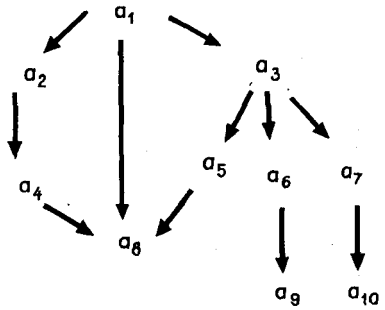
ЗАДАЧИ

V.1. Топологическая сортировка

Следующая задача часто встречается в многочисленных областях. Наиболее классическое приложение – организация сложной работы: одни этапы должны обязательно предшествовать некоторым другим (например, нельзя класть крышу дома, не построив предварительно стен), другие, напротив, могут выполняться в произвольном порядке или одновременно (например, укладка крыши и наружное оборудование). Пытаются найти упорядочение, адекватное различным задачам (возможно, «оптимизированное» по некоторому критерию).

Задачу можно описать так: есть множество A из n элементов a_1, a_2, ..., a_n (в нашем примере – задания). Существует множество C пар [a_i, a_j] («ограничений»); если [a_i, a_j] входит в это множество, говорят, что «a_i предшествует a_j». Так определяемое отношение представляет собой частичное упорядочение, т.е. не существует подмножества A, образованного элементами a, b, c, ..., x, y, такого, что a **предшествует** b**,** b **предшествует** c**,** ..., x **предшествует** y**,** а y **предшествует** a (замкнутая цепь). В частности, никакой элемент не предшествует сам себе, и если a **предшествует** b**,** то b не может предшествовать a.

Можно представить эту ситуацию с помощью ориентированного графа без циклов:



Требуется найти общий порядок, совместимый с заданным отношением, т.е. такое упорядочение элементов $a_{i1}\ a_{i2}, \ldots, a_{in}$ из A что если a_{ij} предшествует a_{ik} то $j < k$

Для вышеприведенного графа решениями являются

$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10}$
$a_1\ a_3\ a_7\ a_{10}\ a_6\ a_9\ a_5\ a_2\ a_4\ a_8$

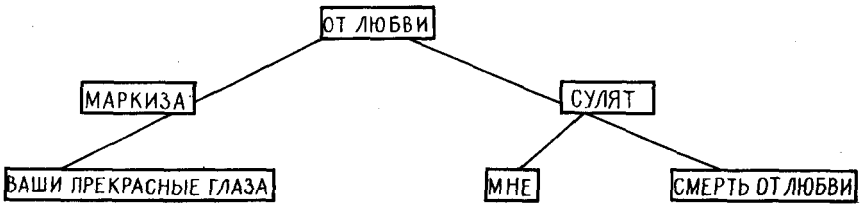
Алгоритм, печатающий упорядоченное решение, описывается в самом схематичном виде:

пока A не пусто **повторять**
| найти элемент из A, например a, такой, что никакой элемент не «предшествует» a, т.е. в C не существует никакой пары вида [x, a];
| **печатать** a;
| извлечь a из A, а также все пары вида [a,x] из C

Требуется доказать корректность этого алгоритма, т.е. что алгоритм заканчивается и никакой элемент a не печатается раньше элемента b, если a **предшествует** b (показать, что оператор «найти элемент из A и т.д.» всегда имеет смысл и выбор всегда возможен, а свойство «C описывает частичный порядок на A» есть инвариант цикла). Обсудите структуры данных, необходимые для использования этого алгоритма.

V.2. Обход дерева влюбленных

Что будет результатом обходов ЛПК, КЛП и ЛПК следующего двоичного дерева[1]:



V.3. Законность *РАВЕНСТВА*

Объясните, почему процедура АЛГОЛа W *EGALITE* (два двоичных дерева; см. V.7.6.1) верна.

[1] Всевозможные перестановки слов в этой фразе обыгрываются в пьесе Мольера «Мещанин во дворянстве». – *Прим. перев.*