ББК 32.973.2-01
Ш90
УДК 681.3.06+519.682

Ошибки программирования и приемы работы на языке ПЛ/1
(новая версия)

Штернберг Леонид Фрицевич

Ш90
Ошибки программирования и приемы работы на языке ПЛ/1.
М.: Машиностроение, 1993. 176 с

В этой книге описаны те нюансы ПЛ/1, которые затрудняют работу программистов. Рассмотрены типичные ошибки, реакция ЭВМ на эти ошибки (что обычно в литературе не описано), показаны некоторые приемы эффективной работы. Книга ориентирована на читателя, имеющего опыт работы на ПЛ/1.

Для широкого круга инженерно-технических работников, программистов, студентов технических вузов.

Оглавление

От редактора новой версии

За полтора десятка лет, прошедших со времени написания этой прекрасной и очень полезной книги, компьютерный мир неузнаваемо изменился. Ушли в прошлое ЕС ЭВМ вместе с их трансляторами ПЛ/1-F, ПЛ/1-О и индексно-последовательными файлами.

Однако, задолго до этого и даже задолго до написания этой книги были сделаны серьезные попытки устранить недостатки языка ПЛ/1, в результате чего Международным и Американским институтами стандартов был принят новый стандарт языка ПЛ/1 X3.74 или «подмножество G». Принятие этого стандарта совпало с началом повсеместного распространения персональных компьютеров и поэтому трансляторы ПЛ/1 на персональных компьютерах появились уже свободными от ограничений и неудачных решений транслятора для IBM 360/ЕС ЭВМ.

Тем программистам, которые перешли на персональные компьютеры, но продолжали работать на языке ПЛ/1, при использовании данной книги стало затруднительным отделить то, на что уже можно не обращать внимания от нужных и полезных советов. А кое-где появились и новые сложности, неведомые во времена ЕС ЭВМ.

Поэтому появилась идея написания новой версии книги. Новая версия сделана следующим образом:

Во-первых, из исходного текста просто убраны те фрагменты, которые рассказывают об особенностях старых трансляторов и о недостатках языка, исчезнувших вместе со старым стандартом «полного» ПЛ/1 ANSI X3.53. Это сократило книгу почти на треть.

Во-вторых, добавлены фрагменты, описывающие особенности стандарта X3.74 и особенности новой реализации языка на персональном компьютере с операционной системой Windows в виде транслятора PL/1-KT.

В некоторых случаях добавлены комментарии к утверждениям автора, если они кажутся спорными.

Все добавления в старый текст отмечены подчеркиванием.

Введение

Большой опыт общения с программистами, работающими над самыми разными классами задач, показывает, что все они сталкиваются примерно с одинаковым кругом проблем, связанных с эффективностью программ и скоростью их написания и отладки, причем очень часто проблемы эти оказываются так и нерешенными. За исключением сравнительно узкого круга профессионалов, досконально знающих машину, язык, систему, основная масса программистов пишет программы, качество которых далеко от оптимального, при этом многие вполне удовлетворены им, так как просто никогда не видели лучшего.

Причины этого таковы.

Во-первых, большинство современных программистов знакомы только с языками высокого уровня, язык машины им не знаком, а потому они совершенно не представляют, что, собственно, делает ЭВМ, когда выполняется тот или иной оператор программы.

Во-вторых, подавляющее большинство книг, в которых описан какой-либо язык программирования, относится скорее к справочникам, чем к учебникам: в них бесстрастно описано большее или меньшее подмножество языка, причем большинство авторов этих книг, видимо, полагает, что чем полнее описан язык, тем лучше. Однако для грамотного программирования нужно не просто знание того, как работает та или иная конструкция языка, а представление о «качестве» этой конструкции: ее эффективности, ошибкоустойчивости и т.д. Дело в том, что точка зрения на трудоемкость отдельных операций у человека и ЭВМ совершенно различна, и одна только информация, «что почем» с точки зрения ЭВМ, заставила бы многих программистов пересмотреть ряд принятых ими решений. Однако почерпнуть эту информацию неоткуда.

В-третьих, во всех книгах по языкам программирования написано, как надо писать программы, и показаны примеры правильных программ, но нигде не анализируются типичные ошибки и их последствия. Такой анализ помогает читателю осознать «коридор возможностей»: так можно и нужно, так еще можно, но это не слишком хорошо, а так уже нельзя.

Наконец, в каждом учебнике по тому или иному языку автор хвалит описываемый язык, сообщая читателю о его достоинствах, но умалчивает о недостатках.

Чтобы исправить сложившуюся ситуацию, нужна книга, которая бы не столько описывала язык, сколько была бы путеводителем по нему: эта конструкция хорошая - ею надо активно пользоваться, а эта - плохая (неэффективна, подвержена ошибкам) и ее лучше вообще не изучать и не использовать; давала бы сведения о том, к каким эффектам приводят те или иные ошибки (грамотная отладка возможна только тогда,

когда программист представляет, чем именно может быть вызван полученный им эффект); наконец, не вникая в подробности работы ЭВМ и внутренний код данных, давала бы информацию о том, насколько трудоемка с точки зрения ЭВМ та или иная языковая конструкция, что позволило бы корректнее использовать средства языка.

Настоящая книга как раз и ставит перед собой цель решить поставленные проблемы и ответить на поставленные вопросы.

Некоторые из них касаются программирования в целом, другие - специфичны для конкретного языка. В качестве языка программирования, который будет подвергнут анализу, взят ПЛ/1. Его выбор обусловлен следующим: этот язык весьма распространен; он очень объемен и, как следствие, содержит огромное поле для непонимания отдельных конструкций и появления ошибок. Опыт показывает, что работающим на нем программистам подобная информация оказывается весьма полезной.

Сколь актуально рассмотрение техники программирования на ПЛ/1, которому уже почти полвека и который был распространен в основном на далеко не новых ЭВМ типа ЕС? Оно актуально по ряду причин. Во-первых, несмотря на свой возраст, язык будет использоваться еще очень долго: помимо его реализации на весьма распространенных ЕС ЭВМ есть реализации на машинах «VAX», «Burroughs», «Эльбрус», «WANG» и персональных компьютерах.

Напомним также, что по многим показателям ПЛ/1 превосходит другие языки. Например, в нем имеются очень удобные средства работы с файлами, обработки прерываний, организации динамических структур и т.д. С появлением новых машин приходят и более новые языки, имеющие на этих ЭВМ хорошие реализации, но программист, работавший на ПЛ/1 и перешедший к работе на модных ныне языках Паскаль, Модула-2, Си, быстро обнаруживает, что ему явно не хватает некоторого привычного языкового сервиса: то, что на ПЛ/1 пишется в одну строку, например динамические массивы, которые заводятся при входе в блок, на этих языках описывается весьма сложно. Так что недостатки ПЛ/1 - это лишь продолжение его достоинств. А перейдя к работе на других языках, программист увидит, что подавляющее большинство описанных выше приемов программирования остается без изменения, вопросы эффективности программ остаются теми же, и некоторое понимание того, как с точки зрения ЭВМ выглядят его решения применить то или иное

средство, всегда поможет ему принять правильное решение (которое не зависит от языка программирования). Так что изучение программирования на языке, имеющем недостатки, не пропадет даром.

Предполагается, что читатель знаком с основами программирования в целом и основными конструкциями ПЛ/1, а также со способом описания синтаксиса конструкции, когда в квадратные скобки берется часть, которая не обязательно должна присутствовать, а в фигурных скобках перечисляются возможные варианты

части конструкции. Предполагается также, что читатель знает, что ЭВМ работает в машинном коде, и имеет представление о трансляции и редактировании связей.

Понятно, что ЭВМ вместе с операционной системой и транслятором выступает как машина, которая как бы понимает текст на языке программирования. Поэтому в ряде случаев будут использоваться выражения вида «ЭВМ делает то-то», не уточняя, делает ли это собственно ЭВМ, транслятор или какая-то другая часть программного обеспечения ЭВМ.

Материал книги в некоторых местах содержит повторы, которые введены в текст умышленно, чтобы облегчить пользование книгой. Например, если какую-то особенность имеет ввод строковых переменных, то об этом упоминается и в главе, посвященной вводу, и в главе, посвященной работе со строками.

Глава 1. СИНТАКСИЧЕСКАЯ И СМЫСЛОВАЯ ПРАВИЛЬНОСТЬ ПРОГРАММЫ

1.1. Синтаксическая диагностика

Общие замечания. Первое, с чем сталкивается программист, начавший отладку программы, - это синтаксические ошибки. Любой транслятор с любого языка при своей работе формирует листинг (на бумаге, на экране, в файле).

Транслятор ПЛ/1 при трансляции печатает текст программы, а при задании специальных режимов - печатает некоторые дополнительные таблицы. При обнаружении ошибок на каждую ошибку выдается диагностическое сообщение; Общий смысл этих сообщений - это «НЕ ПОНИМАЮ», а конкретно каждое диагностическое сообщение дает информацию о каком-то нарушении правил записи.

Наличие сообщений-диагностик на первый взгляд делает поиск синтаксических ошибок весьма простым, но на самом деле некоторые ошибки приводят к правильному с точки зрения ЭВМ тексту, другие ошибки влекут за собой сообщение не на то место, в котором на самом деле допущена ошибка, третьи - помимо сообщения о самой ошибке влекут серию так называемых *цепных диагностических сообщений* на операторы, не содержащие ошибки. Поэтому для быстрого и правильного устранения синтаксических ошибок нужно знание некоторых специальных приемов, которые изложены в этом параграфе. Поскольку разные трансляторы дают разные сообщения, здесь приводятся смысловые описания сообщений, а конкретный их вид может существенно отличаться. Анализ оператора с точки зрения ЭВМ. В операторе-описании

```
DCL A BIN,FIXED, (B,C,D) FLOAT;
```

допущена ошибка: стоит запятая между BIN и FIXED. Но с точки зрения ЭВМ этот оператор абсолютно правильный: в нем описана переменная A с атрибутом BIN (по умолчанию добавится атрибут FLOAT и A получит атрибуты BIN FLOAT, что то же самое, что просто FLOAT), переменная с идентификатором FIXED (напомним, что идентификаторы могут совпадать по написанию со служебными словами) и без атрибутов - по умолчанию она получит атрибуты BIN FIXED. Эта ошибка может пройти вообще незамеченной: ЭВМ ее не заметит, программист ее может заметить только при выполнении программы, когда увидит, что от значений A не отсекается дробная часть (если алгоритм использования переменной A позволит это увидеть). В реализации для персональных компьютеров описание переменной без атрибутов даст диагностическое сообщение «СТРАННО» и присвоит переменной атрибуты BIN FIXED.

Теперь пусть в том же описании допущена другая ошибка (точка с запятой вместо запятой):

```
DCL A BIN FIXED; (B,C,D) FLOAT;
```

С точки зрения ЭВМ в этой строке не один, а два оператора: первый - это правильный оператор-описание переменной A, а второй - ошибочный. Именно на этот второй оператор будет дано на первый взгляд странное сообщение о синтаксической ошибке. Чтобы понять это сообщение надо взглянуть на этот второй оператор, с точки зрения ЭВМ: он начинается с открывающей скобки и сообщение становится понятным.

Замечание. *Еще раз отметим, что приводятся смысловые описания, диагностических сообщений. В разных реализациях диагностики могут отличаться по форме, выдаваться на английском языке. Например, может быть выдано «СИНТАКСИС» или что-либо еще.*

Цепные ошибки. Во многих случаях ошибка в одном операторе помимо сообщения на ошибочный оператор вызывает еще и ряд сообщений на другие (возможно, правильные) операторы, которые могут располагаться и довольно далеко от ошибочного оператора. В программе

```
…PROC ...
DOI=1 TO N; END;END;
…
```

имеется ошибка в операторе DO (пропущен пробел). Естественно, что ЭВМ не может догадаться, что здесь имелся в виду оператор цикла (с ее точки зрения он больше похож на оператор присваивания переменной DOI некоторого ошибочного выражения), поэтому на этот оператор последует сообщение «СИНТАКСИС» (или другое сообщение). Но поскольку ЭВМ не поняла, что это оператор цикла, то первый END она теперь отнесет к оператору PROC; но поскольку за этим END еще продолжается текст программы, то на первый END будет дано цепное сообщение «НЕОЖИДАННЫЙ КОНЕЦ ПРОГРАММЫ» (т. е. END, соответствующий начальному PROC найден раньше конца текста). Понятно, что цепные диагностические сообщения следует игнорировать: они исчезнут вместе с исправлением ошибки. Следует, однако, четко представлять, какое диагностическое сообщение является цепным, ибо в случае, который очень напоминает предыдущий:

```
…PROC...
  DO I=1TO N;
  END; END;
```

(пропущен пробел в том же операторе, но в другом месте) цепного сообщения может и не быть (зависит от транслятора). Хотя на оператор DO будет дано то же сообщение «СИНТАКСИС», но здесь транслятор может понять, что это оператор DO, и сделать правильное сопоставление END.

Некоторые ошибки могут не вызвать диагностического сообщения, но дать цепное сообщение, причем не одно. Например, в программе:

```
…PROC…;
…GOTO L;
BEGIN;
/*     /*
END;
/*    */ …
L: END;
```

вместо признака конца комментария «*/» случайно написали наоборот «/*». С точки зрения ЭВМ здесь ошибки нет: просто за конец комментария она сочтет конец следующего комментария и в качестве комментария воспримет всю отчеркнутую часть. В результате один из операторов END будет воспринят как часть комментария, метка L оказывается внутри блока, а значит, не доступна для оператора GOTO. В результате на оператор GOTO L; получим сообщение «НЕ ОПИСАНО», а на завершающий END - сообщение «НЕОЖИДАННЫЙ КОНЕЦ ФАЙЛА» (т. е. не хватает END для начального PROC).

Наибольшее количество цепных ошибок вызывают ошибки в описаниях. Заметим, что обнаружив ошибку в операторе, транслятор, как правило, прекращает обработку этого оператора, ищет точку с запятой и продолжает работу со следующего оператора. Поэтому при многих ошибках в операторе DCL часть описаний, а именно расположенные за тем символом, где была обнаружена ошибка, остается «незамеченной». В результате этого, часть атрибутов переменных может быть принята транслятором по умолчанию. Если по умолчанию оказались приняты атрибуты переменных, которые должны быть символьными строками, то может последовать серия цепных сообщений «ТРЕБУЕТСЯ ПЕРЕМЕННАЯ» на операторы, где эти переменные используются. Если пропущено описание массива, то запись вида «A(K)» ЭВМ трактует как обращение к функции A с параметром K. Наличие такой записи влечет за собой предупреждение «НЕ ОПИСАНО». Увидев такое сообщение, следует проверить, действительно ли в этом операторе есть обращение к функции, или это цепная ошибка.

Контроль нумерации операторов. Если бы в рассмотренном выше примере с ошибкой в закрывающих символах комментария между комментариями не оказался END, то ЭВМ ошибки не заметила бы: просто она сочла бы часть операторов за комментарий и никаких сообщений не последовало. Для обнаружения ошибок такого рода помимо сообщений надо проверять нумерацию операторов. Любой транслятор как-то печатает нумерацию операторов (например, транслятор PL/1-KT печатает их по ключу «I»). Если вся строка является комментарием или продолжением оператора, начало которого находится в предыдущей строке, то строка остается без номера. Пропуск и нумерации хорошо видны на распечатке - и надо проверить по тексту программы, действительно ли эта строка не должна иметь самостоятельного номера, или это результат ошибки. Наиболее типичные ошибки, в

результате которых прекращается нумерация, - это пропуск закрывающих символов комментария или закрывающего апострофа строки: тогда весь текст далее до встречи закрывающих символов комментария или очередного апострофа транслятор считает продолжением соответственно комментария или строки и не реагирует на точки с запятой (они тоже считаются символами комментария или строки).

Анализ таблицы идентификаторов. Все трансляторы языка ПЛ/1 имеют режимы работы, при которых печатается таблица всех используемых в транслируемой внешней процедуре (головной в том числе) идентификаторов и их атрибутов. Для транслятора PL/1-КТ предусмотрен ключ S. Вообще говоря, таблицы. транслятора программиста интересовать не должны, но некоторые недостатки ПЛ/1 (в частности, наличие принципа умолчания) приводят к тому, что без этой таблицы сложно уяснить себе, как же поняла ЭВМ некоторые детали нашей программы. В таблице идентификаторов все они обычно упорядочены по алфавиту, каждому идентификатору соответствует строка, где указаны: сам идентификатор, возможно, оператор DCL, в котором он, описан, и полный список атрибутов идентификатора. Конкретный вид таблицы зависит от транслятора. Первое, что следует проверить в таблице, это все ли идентификаторы действительно есть в программе, нет ли в таблице «чужих» идентификаторов, появившихся в результате обработки в соответствии с принципом умолчания идентификаторов, набранных с ошибкой или с пропущенными атрибутами. Второе, что следует проверить, это правильность атрибутов каждого идентификатора. В ПЛ/1 имеется множество атрибутов, существенная часть которых в этой книге не рассмотрена, поэтому надо контролировать, чтобы присутствовали все нужные атрибуты и игнорировать незнакомые атрибуты. Читая, например, строку таблицы идентификаторов вида:

```
A (1:100) DECIMAL FLOAT (6)
```

понимаем, что A массив, (от 1 до 100) арифметических (числовых) данных типа FLOAT(6) (игнорируем DECIMAL, если не знаем, что это такое); строка

```
L LABEL CONSTANT
```

означает, что L - метка-константа, т. е. метка, стоящая перед оператором; а строка

```
C ENTRY PARAMETRS(0) FLOAT EXTERNAL
```

означает, что C имеет атрибут ENTRY (т. е. подпрограмма) и выдает значение вида FLOAT.

Поскольку на вывод таблицы идентификаторов расходуются время и бумага, можно рекомендовать выводить ее только на первой трансляции для массового контроля исходного текста и в случаях, когда в программе появляется какая-то ошибка, которую не удается обнаружить.

Независимость трансляции подпрограмм. Каждая подпрограмма обрабатывается транслятором независимо от подпрограмм, которые обрабатывались

раньше или будут обрабатываться позже, и никакая информация об оттранслированной подпрограмме не хранится. Поэтому, если в одной подпрограмме описано, что P - это процедура с тремя параметрами-массивами, а следующая подпрограмма - это как раз процедура P, но с двумя параметрами, которые к тому же оказались не массивами, то транслятор никаких сообщений не даст, так как, во-первых, у него не сохранилось сведений о предыдущей трансляции, во-вторых, ниоткуда не следует, что эта подпрограмма P является частью той же программы, в которую входит и предыдущая подпрограмма.

1.2. Диагностика ошибок времени выполнения

Классификация ошибок. Если программа синтаксически правильна, то трансляция проходит успешно, и далее ЭВМ переходит к собственно выполнению программы. Но абсолютно понятная транслятору конструкция может оказаться невыполнимой вообще или в конкретной ситуации (при конкретных данных), что приводит к ошибкам времени выполнения, которые можно разделить на два вида:

- программа благополучно завершает свою работу, выдав некоторые результаты, которые ни на что не похожи или же (худший вариант) похожи, но неверны;
- при выполнении программы возникает ситуация, когда продолжение работы невозможно, и происходит аварийное завершение программы.

Первый случай находится полностью в компетенции, программиста, ибо с точки зрения ЭВМ программа верна. Второй же случай также можно разделить на три подслучая:

- возникает ошибочная ситуация, предусмотренная в языке ПЛ/1 (переполнение, деление на нуль, выход индекса за границу массива и т. д.) - в этом случае получаем сообщение времени выполнения, в котором в терминах ПЛ/1 дается описание ситуации и указывается место, где она произошла;
- возникает ситуация, которая ошибочна с точки зрения ЭВМ (точнее, операционной системы), а с точки зрения языка ПЛ/1 вообще быть не может, например, от ЭВМ требуется выполнение несуществующей команды или обработать данные, находящиеся вне разрешенных адресов памяти ЭВМ, - в этом случае выдается сообщение ПЛ/1 с некоторыми терминами, которые непонятны человеку, не знающему особенностей конструкции ЭВМ;
- возникает ситуация, которая не является ошибкой ни с точки зрения ПЛ/1, ни с точки зрения операционной системы, но в конкретных обстоятельствах невыполнима, например, программа требует слишком большого объема оперативной памяти или при записи файла на диск исчерпано отведенное

файлу место. В этом случае программа прекращает работу с диагностическим сообщением операционной системы, а не с диагностическим сообщением ПЛ/1.

Диагностика машинных ошибок. Ошибка с диагностическим сообщением ПЛ/1 не требует комментариев, ибо причины возникновения ситуаций ПЛ/1 описаны в учебниках по языку. В случае же машинных ошибок программист может встретить в сообщениях термины, которые нуждаются в переводе с машинного на русский. Рассмотрим эти термины и их смысл.

Защита памяти («НЕВЕРНЫЙ ДОСТУП») - попытка работать за пределами памяти, отведенной нашей программе. Эта ситуация возникает, если в результате ошибок была сделана попытка «залезть» в «чужую» память, принадлежащую операционной системе или другой программе. Может возникнуть при выходе индексов за пределы строки или массива, чаще всего - при значительном выходе (т. е., например, в массиве из 10 элементов пытаемся добраться до 10000-го элемента). Этот случай можно заблокировать включением ситуаций STRINGRANGE и SUBSCRIPTRANGE и получить вместо этого сообщения нормальную ситуацию ПЛ/1 (в трансляторе PL/1-КТ – включить контроль режимом «Т»). Можно получить это сообщение и при ошибках в работе с указателями, связанными с тем, что указатель адресует часть памяти, которая уже освобождена оператором FREE, но возможны и другие ошибки значений указателей. К сожалению, никакими программными средствами заблокировать эти ошибки нельзя. Наконец, это сообщение может возникнуть, если обратиться к процедуре и не передать ей параметр, тогда процедура пытается взять значение параметра из случайного места памяти с возможной попыткой «залезть за память». Подробнее этот случай описан в гл. 6, заблокировать его программными средствами также нельзя.

Операция («ПРИВИЛИГЕРОВАННАЯ КОМАНДА»). Это попытка выполнить операцию, которая разрешена только программам операционной системы. Естественно, эта операция не может появиться в нашей программе. Эта ошибка означает, что была сделана попытка выполнить в качестве части программы что-то, что частью программы не является. Она возникает практически в двух случаях: в качестве фактического параметра подпрограммы, который должен быть именем другой подпрограммы, передан параметр другого типа или не передано вообще ничего, а также при обращении к несуществующей подпрограмме. Еще один частый случай – разрушение системного стека при неверном описании процедур и функций.

Данные. Это значит, что содержащиеся в памяти значения некорректны. Эта ошибка может возникать только при работе с данными типа DEC FIXED, если эти

данные получали значение не прямым присваиванием, т. е. в ситуациях, когда контроль типов и преобразования типов сработать не могут (передача данных через параметры, использование функции UNSPEC как псевдопеременной, чтение из файла).

Запрос слишком большого объема оперативной памяти. Может возникнуть по следующим причинам:

- слишком большой объем памяти запрашивается по оператору ALLOCATE;
- в результате ошибки в организации программы получилась слишком большая глубина рекурсивных вызовов процедур.

1.3. Понятие «эффект не определен»

Смысл термина. В документации ПЛ/1 описан ряд моментов, о которых говорится, что при их возникновении эффект работы программы не определен. Что означают эти слова? Во-первых, это значит, что транслятор эти моменты не фиксирует и диагностического сообщения не выдает, во-вторых, при выполнении программы эти ситуации также зафиксировать невозможно, следовательно, программа продолжает работу, но результат в этом случае не оговорен, т.е. в описании языка не описано, что в этом случае должно получиться. Возможны несколько вариантов работы программы в таких ситуациях.

Первый вариант: транслятор в подобных случаях делает что-то определенное, в итоге получается программа, которая выполняет вполне определенные действия. Можно выяснить, как поступает транслятор, и определить эффект программы, однако при смене транслятора (или даже версии транслятора) возможно изменение эффекта программы.

Второй вариант: транслятор не предпринимает никаких действий, и выполнение программы зависит от операционной обстановки, т.е. от того, что происходит на ЭВМ в текущий момент или что происходило ранее. В итоге одна и та же программа может выдавать разные результаты при разных ее выполнениях.

Примером первого варианта может служить работа с выключенной ситуацией OVERFLOW: в этом случае результат не определен в языке, но ЭВМ все же дает какой-то результат. Понятно, что ЭВМ другой модели может дать иной результат. Примером второго варианта является работа с переменной, которой не было присвоено значение. Если транслятор не чистит память, т.е. не рассылает в отведенную переменным память какие-то стандартные значения или работа идет с указателями, то при выполнении программы в качестве значения переменной будет взято то значение, которое осталось в этом месте памяти от выполнения предыдущей программы. Понятно, что при разных выполнениях одной и той же программы там могут оказываться разные значения.

Ясно, что ситуации, когда эффект не определен, ставят проблемы в отладке, для успешного преодоления которых надо знать, какие именно причины могут дать неопределенный эффект, и какие именно последствия они могут вызвать.

Не присвоенные значения. Наиболее частый случай, дающий неопределенный эффект, - это работа с переменными, не имеющими значений. Последствия этого весьма разнообразны. Первый признак наличия такой ситуации - это разные результаты, выдаваемые программой при разных выполнениях на неизменных исходных данных. Но это возникает в основном при использовании переменной без значения в вычислениях. Если же целое значение переменной используется в

качестве индекса, то имеют место эффекты, связанные с выходом индексов за границу массива. Наконец, в некоторых случаях возникает ситуация «данные», подробно описанная в п. 6.2. Это ситуация означает, что переменная содержит значение, которое она содержать не может, и также является признаком работы с переменной без значения. Эта ситуация может возникать только при работе с данными, имеющими атрибут DEC FIXED - способ кодирования данных для обычных числовых переменных с атрибутами BIN FIXED или FLOAT таков, что любое содержимое памяти ЭВМ может быть как-то проинтерпретировано.

Выход индекса за границы массива или строки. Если не включена контролирующая ситуация SUBSCRIPTRANGE или STRINGRANGE (ERROR(16) и ERROR(19) соответственно в трансляторе PL/1-КТ по ключу Т), то контроль выхода индекса за допустимые границы не ведется. В итоге делается попытка выполнить обработку элемента массива или символа строки, лежащего за пределами памяти, отведенными этому массиву или строке. Далее возможны следующие случаи.

Иногда обработка проходит внешне нормально, т.е. диагностических сообщений ЭВМ не выдает. Если элемент считывается, то получаем работу с некоторым элементом, не имеющим значения, со всеми описанными выше эффектами. Если элемент записывается, то может быть испорчено значение той переменной, которая занимает память, расположенную рядом массивом (строкой) за пределы которого мы «залезли». Непосредственно операция записи выполняется нормально, а далее при использовании испорченной переменной возникают описанные выше эффекты. При записи «за массив» или «за строку» не обязательно будет испорчено значение какой-то переменной - может быть испорчена какая-то служебная информация, например, адрес возврата из подпрограммы: в этом случае странные диагностические сообщения возникают при исполнении каких-то, на первый взгляд, совершенно «безобидных» операторов, например, RETURN.

Описанные эффекты возникают, если при выходе индексов за допустимые диапазоны, происходит работа с элементом памяти, принадлежащим нашей программе. Но неправильное значение индекса может быть таково, что элемент с таким индексом должен бы располагаться за пределами «нашей» (отведенной программе) памяти - либо в памяти, принадлежащей другой программе (в частности, самой операционной системе). Возникает ситуация, называемая «защита памяти».

Аналогия. *Предположим, имеется строй школьников, сгруппированный по классам. Если в классе 25 человек, а мы пытаемся добраться до 35-го человека путем отсчета нужного числа людей от первого человека данного класса, то мы либо доберемся до 10-го человека из соседнего класса, что может пройти незамеченным, либо, если класс последний в строю, можем попасть на стоящих*

рядом со строем учителей и без труда понять, что попали куда-то не туда («защита памяти»).

Несоответствие типов параметров или данных в файле. ПЛ/1 не ведет контроль за соответствием типов формальных и фактических параметров процедур, поэтому при несовпадении типов возможны различные странные эффекты, связанные с тем, что ЭВМ пытается расшифровать по одним правилам то, что зашифровано на самом деле по другим правилам. Подробнее об этом сказано в гл. 6. В частности, здесь возможны сообщения «данные».

Работа с некорректными указателями. При работе с указателями ЭВМ не может выполнять почти никакого контроля, поэтому при ошибках в работе с указателями возможны все перечисленные выше эффекты, ибо ничто не мешает указателю показывать «в никуда», «вообще не туда» или на данные не того типа, который имел в виду программист.

Краткие итоги. Суммируя сказанное выше, получаем следующие возможные причины странного поведения программы. Разные результаты при неизменной программе и данных - наличие переменных без значений либо работа не со «своей» памятью из-за выхода индексов за границы массивов (строк) или ошибок в указателях.

Ошибка «данные» - неправильный тип параметров процедур, неправильное чтение из файла (неправильный тип данных в читаемой переменной).

Ошибка «защита памяти» - выход индексов за границы, ошибки в указателях.

1.4. Правильность и эффективность программы

Постановка проблемы. Если программа дает неправильные результаты, то ее скорость или занимаемая ею память совершенно несущественны. Но если программа дает правильные результаты, однако требует для этого столько времени, сколько не может себе позволить ее пользователь, или столько памяти, сколько нет на доступной ЭВМ, то различие между наличием правильной программы и ее отсутствием практически сводится к нулю. Итак, об эффективности программы надо думать. Во многих случаях программисты не обращают внимания на эффективность программы: работает программа два часа - значит, будем тратить два часа. В основном это происходит потому, что программист не знает, чем можно повысить эффективность программы, уповая в лучшем случае на оптимизирующий транслятор. Но оптимизирующий транслятор не может принять за программиста *стратегическое* решение - он может только *технически* наилучшим способом реализовать те' идеи, которые заложил в программу ее автор. В частности, оптимизирующий транслятор не может заменить типы описанных в программе переменных, а правильный выбор типов данных может резко повысить эффективность программы.

Чтобы получать эффективные программы, необходимо представлять «что почем» с точки зрения ЭВМ, ибо точка зрения ЭВМ на трудоемкость отдельных операций в корне отличается от точки зрения человека. Причем эта трудоемкость, как и затраты памяти, зависит от типов обрабатываемых данных. Для принятия правильных решений нет никакой необходимости знать форму представления данных в памяти ЭВМ и точные скорости операций - эти данные зависят от модели ЭВМ, транслятора и прочего. Достаточно знать относительные характеристики, т. е. что больше, что меньше; а относительные характеристики существенно меньше зависят от ЭВМ и транслятора и практически постоянны.

Трудоемкость операций. Измерять трудоемкость операций можно временем их выполнения и числом команд машины, которые требуются на эту операцию (последнее определяет длину программы, получающуюся после трансляции). За единицу времени примем время выполнения операции сложения целых чисел, а за единицу памяти одно машинное слово, т. е. память, достаточную для хранения действительного или целого числа (4 байта).

Приблизительные характеристики операций приведены в табл. 1.1.

Расход памяти. Для оценки принятых решений с точки зрения затрат памяти, нужно иметь в виду следующие моменты.> Память расходуется, во-первых, на собственно данные, во-вторых, на различную служебную информацию. В-третьих, из-за выравнивания для ускорения выполнения программы часть памяти просто

теряется. Затраты памяти для собственно данных показаны в табл. 1.2 в первой ее части. Во второй части показаны затраты памяти для служебной информации.

Например, для файла помимо значений в памяти размещается еще так называемый дескриптор (описатель, паспорт) файла, содержащий различные сведения, на него-то и требуется дополнительная память.

Таблица 1.1 Приблизительные относительные характеристики операций

Операция	Время выполнения ручной счет (3-зн.числа)	Время выполнения микрокалькулятором	Время выполнения ЭВМ	Затраты памяти на команды в словах [1*]
Сложение целых	1	1	1	1
Вычитание целых	1.3	1	1	1
Сравнение целых	d [2*]	d	1	1
Умножение целых	8	1	2	1
Деление целых	10	1	3	1
Сложение действительных	1+e [3*]	1	2	1
Вычитание действительных	1.3	1	2	1
Сравнение действительных	d	d	2	1
Умножение действительных	8	1	3	1
Деление действительных	10	1	4	1
Индексирование	0	0	1+2k [4*]	2
N-кратное индексирование	0	0	3n-2+2k	2n
Циклирование WHILE-цикл [5*]	e**2	e**2	2	3
Циклирование BY-TO-цикл [6*]	e**2	e**2	5…8	7…8
Присваивание (запись на бумагу)	d	d	2	2
Конкатенация строк	-	-	6…10	3…6
Логические операции «и/или»	-	-	0.5…2	1…3
Ветвление IF...THEN...ELSE	-	-	1	2
Переход GO TO	e**2	e**2	0.5	1
Обращение к параметру процедуры	-	-	1	1

Вызов процедуры или функции	-	-	Много 7*	Много
Преобразование целое – действительное	-	-	10…12	10…12

Примечания:

1* Слово = 4 байта

2* Выполняется вручную, зависит от личных качеств расчетчика.

3* е означает «очень мало», е**2 – «пренебрежимо мало> по» сравнению со временем операции сложения.

4* к=0.5 для индексации числовых массивов, к=1 для индексации строковых массивов.

5* Без учета затрат на вычисление выражения, стоящего в условии.

6* В предположении, что параметр цикла целый. Если шаг и конечное значение заданы выражениями, то они вычисляются один раз перед началом цикла, поэтому затраты на них не учтены.

7* Означает «очень много» - точные значения зависят от ЭВМ, транслятора, типа параметров и т. д.

Таблица 1.2. Затраты памяти на хранение данных разных типов

Атрибуты переменных	Объем памяти в байтах на одно значение
BINARY FIXED(p) p<16	2
BINARY FIXED(p) p>16	4
DEC FIXED(p,q)	(p+q+2)/2
FLOAT DECIMAL(p) p<6	4
FLOAT DECIMAL(p) 6<p<16	8
CHARACTER(k)	K
BIT(k)	(k+7)/8
VARYING	Дополнительный байт на каждое значение
файл	30-40 байт на паспорт и заданное количество байт на буфер

Числовые данные типа BINARY FIXED и FLOAT для ускорения доступа могут требовать так называемого выравнивания, т. е. могут размещаться, начиная не с любого байта, что приводит к потерям памяти. Потери памяти происходят и при размещении массивов типа BIT: хотя элемент и не занимает полный байт, но

следующий элемент размещается с начала нового байта. Поэтому массив с атрибутами (1:m) BIT(n) будет занимать m[(n+7)/8], а не [(mn+7)/ 8] байт, хотя очевидно, что второе число может оказаться значительно меньше первого.

Понятно, что в основном расход памяти существен для массивов, в особенности больших размеров, но и в малых программах нецелесообразно тратить память зря.

Оценка принимаемых решений. Разумеется, не предполагается, что программист по приведенным таблицам будет вычислять скорость программы и ее размер, но взгляд на таблицы позволяет сделать ряд выводов. Во-первых, можно выстроить последовательность атрибутов

BIT - BIN FIXED - FLOAT - DEC FIXED

и сформулировать общее правило (имеющее, правда. некоторые оговорки): каждый следующий тип требует больше памяти и больше времени на обработку значения. Поэтому, если есть выбор, то нужно брать возможно более левый тип из этого ряда. Для признака, принимающего значения «да/нет» эффективнее атрибут BIT(1), а не переменная типа BIN FIXED, принимающая значения 0 и 1; для индекса или счетчика наилучший атрибут BIN FIXED и т. д. Эти же таблицы показывают, что нецелесообразно использовать маленькие массивы: четыре переменные К1, К2, КЗ, К4 займут вдвое (если не более) меньше памяти, чем массив К (1:4), и их обработка мелкими циклами типа

```
DO 1=1 TO 4; K(I)=10*I; END;
```

по всем параметрам уступает простой серии присваиваний

```
К1=10; К2=20; КЗ=30; К4=40;
```

поскольку много времени требует операция циклирования. Если имеется выбор между двумя одномерными массивами DCL (N1,N2)(1:L) ... и двумерным массивом DCL N(1:2,1:L) ... то предпочтение следует отдать первому варианту: двойная индексация при обработке двумерного массива даст заметные потери в скорости программы.

И вообще индексацию надо использовать только там, где без нее обойтись нельзя.

Там, где надо использовать вложенные циклы и порядок циклов безразличен с точки зрения алгоритма, варианты, отличающиеся только порядком циклов

```
DO I=1 TO N; DO J=1 TO M; ... END; END;
DO J=1 TO M; DO I=1 TO N; ... END; END;
```

различаются по скорости, ибо в первом случае потребуется M*N+N операций циклирования, а во втором - M*N+M.

Эффективность языковых конструкций. Очень часто одно и то же можно запрограммировать несколькими способами - как в таких случаях выбрать оптимальный? Разумеется, можно прикинуть по приведенным выше таблицам трудоемкость используемых операций, но оценку можно выполнить и другим

способом, руководствуясь общим правилом: *специализированная конструкция работает всегда эффективнее общей.*

Простейший пример - как эффективнее вычислить квадратный корень:

```
A=B**(1/2);
A=B*-0.5; или
A=SQRT(B);
```

Понятно, что первый вариант уступает второму, так как 1/2 - это операция, которую машина будет выполнять, а значит, тратить время. Сравнение же второго и третьего вариантов можно сделать по общим правилам: возведение в степень предназначено для возведения в любую степень (общая конструкция), а функция SQRT - только для возведения в степень 0.5 (специализированная конструкция), следовательно, последний вариант эффективнее. (Общий способ возведения в степень сводится к вычислению двух функций: A**B приводится к вычислению EXP(LOG(A)*B).)

Неужели машина (точнее, транслятор) не видит, что 1/2 - эта то же самое, что и 0.5, и что возведение в степень 0.5 - это извлечение корня? Все дело в том, какой транслятор. Так называемые оптимизирующие трансляторы подобные вещи обнаруживают (затратив время на дополнительный анализ) и могут реализовать все три варианта одинаково, наилучшим образом, сведя их к последнему варианту. Но программист должен понимать, какой вариант лучше. Можно сказать, что третий вариант как *языковая конструкция* лучше, т. е. она заведомо даст не худшие результаты, чем две другие, независимо от транслятора.

Следует помнить, что обращение к элементам массива (операция индексирования) - это для ЭВМ операция более тяжелая, чем умножение, поэтому не стоит без необходимости группировать переменные в массивы.

Можно сказать, что группировать переменные в массивы целесообразно тогда, когда они все требуют одинаковой обработки и их достаточно много.

Общие и специализированные конструкции. Чтобы понять, какая конструкция более общая, а какая - более специализированная, надо определить, какая из них корректно сработает на большем множестве данных. Например, сравним два способа организации Цикла:

```
DO X=A BY H TO B; ... END;
```

и

```
DO X=A BY H WHILE(X<=B); ... END;
```

Первый вариант корректно сработает при любых значениях A, B и H (при ненулевом H), второй защитится, если знак H не совпадает со знаком (B-A), следовательно, первая конструкция более общая, а потому менее эффективная. И действительно, это заключение, сделанное из общих соображений, подтверждается анализом того, как работают эти циклы. Первый вариант проигрывает в скорости из-

за того, что на каждом шаге выполняется более сложная проверка знак(H)=знак(B-X)? или, что то же самое, H*(B-X)>0?.

Разумеется, в подавляющем большинстве случаев эта разница непринципиальна и программисту можно рекомендовать первым вариант именно потому, что он не зацикливается, но есть и случаи, когда от программы надо добиться максимальной скорости, и программист должен знать, где искать резервы.

Глава 2. ОСНОВНЫЕ ЯЗЫКОВЫЕ КОНСТРУКЦИИ

2.1. Описания типов

Числовые типы. Для описания числовых переменных в языках программирования обычно имеются атрибуты «целый» и «действительный». В ПЛ/1 программисту предоставляется «богатый» выбор: можно управлять системой счисления, в которой представляется число, точностью, формой представления и т. д. В результате программист запутывается и выбирает совсем не оптимальные атрибуты. «Зачем выбирать двоичную систему счисления, если мне удобнее работать в десятичной?» - думает программист и выбирает десятичную, тем самым нередко замедляя свою программу раз в шесть.

На самом деле для правильного описания числовой переменной надо определить, для чего она будет использоваться, и исходить из смысла задачи, а не из смысла служебных слов ПЛ/1.

В программировании устоялись понятия *целого* значения и *действительного* значения. Принципиальная разница между ними в том, что *целые значения - точные*, а *действительные - приближенные* (в программировании целые значения не являются подмножеством действительных). Эквивалентом того, что в других языках называется целым значением, в ПЛ/1 является значение с атрибутами BINARY FIXED. Не надо обращать внимание на слово BINARY: вопрос, в какой системе счисления хранит машина значения, не должен волновать программиста. Словосочетание «BINARY FIXED» надо запомнить как заклинание и использовать как атрибут для описания переменной, предназначенной для хранения целых чисел. Аналогично для описания действительных переменных будем использовать атрибут FLOAT.

Вопросы, связанные с атрибутами точности, отложим до главы 5. Пока же отметим, что вне зависимости от атрибута BINARY или DECIMAL и заданной точности все переменные с атрибутами FLOAT(p) хранятся одинаково, если $p<6$ при атрибуте DECIMAL или отсутствии атрибута системы счисления вообще либо если $p<24$ при атрибуте BINARY, поэтому достаточно только атрибута FLOAT.

Переменные же с атрибутами DEC FIXED - это уже совершенно иной способ хранения, ориентированный на экономические задачи, где много ввода-вывода и мало счета. Такие числа в арифметических операциях обрабатываются примерно в шесть раз медленнее, чем числа BINARY FIXED, а если они имеют более трех цифр, то и памяти занимают больше. Таким образом, в обычных расчетных задачах, а также в экономических задачах для внутренних данных, не подлежащих вводу-выводу (например, индексов, параметров циклов), следует использовать атрибут BIN FIXED.

Выводы. Для описания числовых переменных следует использовать атрибуты BIN FIXED и FLOAT. Только в экономических задачах следует использовать атрибуты DEC FIXED, впрочем, можно обойтись без таких атрибутов вообще.

Использование десятичной арифметики. Для экономических задач характерна обработка больших объемов информации, причем вычислительные операции - это небольшая часть работы, а существенная часть работы приходится на ввод-вывод. В этой ситуации невыгодно хранить данные в двоичном представлении, так как тогда много времени будет отнимать перевод чисел в двоичное представление при вводе и обратно в десятичное при выводе, что перекроет всю экономию от ускорения вычислительных операций. Поэтому в экономических программах рекомендуется использовать описатели DEC FIXED. Однако это не означает, что так надо описывать все данные - те из них, которые не предназначены для ввода-вывода, а используются только внутри программы, лучше описать BIN FIXED. Например, если в программе организуете цикл

```
DO I=1 TO N;
...A(I)...
END;
```

то параметр цикла I следует описать именно как BIN FIXED: он используется только внутри программы и никогда не будет переводиться в десятичное представление, а для использования его в качестве индексов для ЭВМ удобнее именно такие атрибуты. Описатель DEC FIXED замедлит работу, а если программист решил еще и задать малую разрядность, скажем, DEC FIXED(2,0), то при N=100 получим ещё и зацикливание: при присваивании значения 100 переменной с атрибутами DEC FIXED(2,0) третья слева цифра будет отсекаться и переменная получит значение нуль.

Выбирая атрибуты для внутренних переменных, следует учитывать, с какими переменными она будет взаимодействовать в программе. Например, упомянутая выше переменная «I» взаимодействовала в основном сама с собой (увеличивалась на единицу) и использовалась в качестве индекса, поэтому ей следовало дать самые удобные для ЭВМ атрибуты. Но если некая переменная, не предназначенная сама для вывода (например, в ней накапливается сумма, которая затем используется для каких-то дальнейших расчетов), часто взаимодействует с другими переменными, которые предназначены в основном для вывода и потому имеют атрибуты DEC FIXED, то и этой переменной следует дать такие же атрибуты, ибо в противном случае при каждой операции с этой переменной один из операндов будет приводиться к форме другого.

Иными словами, старайтесь, чтобы в операциях участвовали операнды с одинаковой формой представления, - это ускоряет программу.

И хотя для экономических данных рекомендуется все же использовать данные с атрибутами DEC FIXED(p,q), программист должен помнить, что таких данные действительно имеют разное представление в памяти в зависимости от p и q, для них действительно выполняются все правила ПЛ/1 относительно точности результата. В итоге иногда возникают неожиданные эффекты. Классический пример - это выражение 25+1/3, которое дает переполнение, ибо результат выражения 1/3 имеет атрибуты DEC FIXED(15,14), а 25 имеет атрибуты DEC FIXED(2,0). Итоговое значение должно получить атрибуты DEC FIXED(16,14), но поскольку максимальное значение p может быть только 15, то получаются атрибуты DEC FIXED(15,14) и значение 25.33333333333333 не помещается в отведенные ему разряды. Точно такой же эффект получится и при других данных в выражении A+B/C при описаниях

```
DCL A DEC FIXED(2,0), (B.C) DEC FIXED(1,0);
```

если A имеет значение большее или равное 10. Теоретически подобные эффекты возможны и с данными других типов, но реализация ПЛ/1 одинаково хранит данные. как с атрибутами FLOAT(3), так и с атрибутами FLOAT(5), а потому не выполняет все предписания ПЛ/1 относительно точности результата. В итоге с этими типами данных подобных эффектов не возникает.

Принцип умолчания. Позаимствованный языком ПЛ/1 из языка ФОРТРАН принцип умолчания, согласно которому все неописанные переменные получают атрибуты BIN FIXED или FLOAT в зависимости от первой буквы идентификатора, с современных позиций считается крайне вредным, так как выключает часть синтаксического контроля. В стандарте языка X3.74 неописанные переменные не допускаются.

Принцип умолчания был введен в ПЛ/1 для обеспечения удобства программисту, чтобы он не тратил время на написание лишних слов. С современных позиций эта экономия напоминает экономию на противопожарных средствах: на поиске одной ошибки теряется вся экономия времени, накопленная за год. Принцип умолчания в ПЛ/1 весьма развит и позволяет опускать довольно много описателей, однако внешняя его логичность иногда дает неожиданные эффекты, ибо транслятор добавляет недостающие атрибуты по не всегда очевидным правилам.

Рекомендации по использованию принципа умолчания таковы.

Разумеется, всегда выписывать все атрибуты переменной нет никакой возможности, ибо их очень много:

```
DCL N DEC FLOAT (6) AUTOMATIC INTERNAL;
```

Однако часть из них по умолчанию стандартна, а другая зависит от других атрибутов. Опускание атрибутов первой группы безопасно: и программист и транслятор вспоминают о них только тогда, когда эти атрибуты нестандартны. Опускание же атрибутов из второй группы чревато ошибками, и рекомендация по

этой части принципа умолчания однозначна: не пользоваться никогда. Заметим, что отказ от использования принципа умолчания и привычка все описывать не спасают программиста от ошибок, которые возникают из-за опечаток в наборе программы, но все же заметно снижают вероятность ошибок.

Синтаксическое неединообразие. Некоторую опасность для программиста представляет нарушение в ПЛ/1 синтаксического единообразия в разных конструкциях. Например, описание простой переменной с присваиванием ей начального значения

```
DCL A BIN FIXED INIT(1);
```

и описание той же переменной без начального присваивания, но с присваиванием значения отдельным оператором

```
DCL A BIN FIXED; ... A=1;
```

эквивалентны. Однако если взять аналогичные операторы, но уже с массивом:

```
DCL A(10) BIN FIXED INIT(1);
DCL A(10) BIN FIXED; ...  DO I=1 TO 120; A(I)=1; END;
```

то они не эквивалентны: первый оператор присвоит значение только первому элементу массива, а второй - каждому элементу массива.

Если же надо, чтобы все элементы массива получили одинаковые начальные значения начальными присваиваниями, то надо написать так:

```
DCL A(10) BIN FIXED INIT((10)1);
```

2.2. Выражения и присваивания

Преобразования при присваивании. Каждая переменная может принимать значения только того типа, который она имеет по описанию или по умолчанию. Если ей присваивается значение другого типа, то это значение нужно сначала преобразовать к типу переменной. Действительное значение преобразуется к целому типу путем отсечения дробной части (не округление, а отсечение!).

Целое число при преобразовании к действительному виду становится приближенным. Например, операторы

```
DCL K BIN; K=1; K=1.25; K=1.9;
```

все присвоят целой переменной К значение, равное единице.

Если К целая переменная, а А - действительная, то значение К может изменить пара операторов А=К; К=А; ибо если К имела значение 20, то после первого присваивания получит значение, лишь приближенно равное 20, и если оно равно 19.999..., то после второго присваивания К получит значение 19. На самом деле такой эффект на малых числах не возникает, он проявляется на числах, больших чем 16**6; переменная, содержащая такое число, должна быть описана с атрибутом BIN FIXED(31).

Автоматические преобразования типов. Другой опасностью является автоматическое преобразование типов, которое действует не только при присваивании, но и в выражениях. Например, пусть выражении A^=B+C программист ошибся и написал A=^B+C получается, что операция «не» применена к числовой переменно B. Совершенно очевидно, что это ошибка: что такое «не 5»? 6? Или 3.44? Однако числовое значение B по довольно сложным правилам преобразуется в битовую строку, к каждому биту которой применяется операция «не», в результате получается новая битовая строка. Но для сложения с C нужно число, поэтому битовая строка по столь же сложным правилам преобразуется в число, которое складывается с C, и результат сравнивается на равенство с A. Естественно, что почти всегда получается ответ «не равно». Таким образом, наличие принципа умолчания и автоматического преобразования типов выключает существенную часть синтаксического контроля и возлагает на программиста контроль за описками. Разумеется, это плохое качество языка, но, к сожалению, единственное, чем можно помочь программисту, это предупредить его о необходимости повышенного внимания.

Автоматическое преобразование типов способствует и проявлению некоторых ошибок преемственности, т. е. ситуаций, когда конструкция языка похожа на конструкцию другого языка, но работает не так. Например, программист, ранее работавший на языке АЛГОЛ-60 или Си, знаком с множественным присваиванием и знает, что есть такая возможность и в ПЛ/1. Однако, если он по привычке напишет I=J=1; предполагая, что как I, так и J при этом получат значение, равное единице, то его ждет ловушка: на ПЛ/1 такая запись означает, что I получит значение выражения J=1, т. е. первый знак «=» это присваивание, а второй - сравнение. Разумеется, это ошибка, ибо сделана попытка присвоения числовой переменной логического значения, но автоматическое преобразование типов не позволит сразу предупредить программиста об этом: логическое значение, которое скорее всего равно '0'B, будет преобразовано в числовое значение и I получит значение 0, а затем программисту придется искать ошибку. Транслятор PL/1-КТ в подобных случаях дает предупреждение «СТРАННО».

Наиболее распространенная ошибка из этой серии - это проверка принадлежности значения X отрезку [A,B], которую пытаются выполнить так: IF A<=X<=B THEN ...

Здесь будет выполнена проверка A<=X, которая даст ответ '1'B или '0'B, а затем это логическое (битовое) значение будет сравниваться с B: вместо сообщения об ошибке это значение преобразуется к числовому типу и сравнение дает трудно предсказуемый результат - во всяком случае не тот, который ожидается.

Приоритеты операций. На первый взгляд приоритеты операций в ПЛ/1 выглядят достаточно логично: сначала выполняются унарные (одноместные, с одним

операндом) операции, затем бинарные арифметические, потом сравнения. Но в этой логике есть один дефект: к унарным операциям относится и логическая операция «^», которая по приоритету выполняется раньше арифметических.

В итоге в выражении «^A+B>C», где имелось в виду ^(A+B>0), операции будут выполняться в порядке ((^(A)+B)>C). Далее срабатывают автоматические преобразования типов и никаких сообщений ЭВМ не выдаст, однако результат будет трудно понимаемым.

Разумеется, в данном примере можно было бы написать «A+C<=B» и проблем бы не было, но что делать, если программист по каким-то причинам написал так, как показано в первом варианте?

Учет атрибутов переменных. Во всех языках программирования имеются правила определения атрибутов результата в зависимости от атрибутов участвующих в них операндов. В некоторых языках эти правила минимальны: нельзя использовать операнды с разным атрибутами и, следовательно, нет проблем, так как все преобразования типов программист должен указывать явно. В ПЛ/1 определены автоматические преобразования почти любого типа в любой другой и транслятор аккуратно подсчитывает атрибуты точности в каждой операции. В итоге возможны описанные выше эффекты с вычислением выражений вида «25+1/3». Возникновение переполнения - это не худший вариант, так как программист получает диагностическое сообщение, худший вариант - это «молчаливое» отсечение разрядов, которое трудно найти.

Чтобы избежать проблем, связанных с этой особенностью языка ПЛ/1, можно дать две рекомендации. Во-первых, не используйте программах константы, участвующие в операциях умножения деления вместе с числами типа FLOAT, ибо с точки зрения ПЛ/ записанные в десятичном виде константы без буквы «E» и порядка имеют атрибуты DEC FIXED, а следовательно, начинаются преобразования типов и подсчет точности и разрядности со всем вытекающими отсюда последствиями. Вместо констант надо использовать переменные с нужными атрибутами, которым присвоено начальное значение, - тогда оно сразу получает нужные атрибуты. То есть вместо записи вида

```
DCL (A.B) FLOAT,...
B=A/20.5;
```

лучше писать

```
DCL (A,B)FLOAT, K FLOAT INIT(20.5),... B=A/K;
```

Во-вторых, в сомнительных случаях используйте функции явного приписывания атрибутов FIXED и FLOAT: B=25+FIXED(A/3,7,3);

Это означает, что результат операции A/3 должен получиться с атрибутами FIXED(7,3).

Есть и более простое правило: пишите все константы в научной нотации, ошибки не будет, например:

```
DCL (A.B) FLOAT,...
B=A/20.5E0;
```

2.3.Ветвления

Особенности синтаксиса. Условный оператор ПЛ/1 имеет один крупный недостаток, создающий трудности в отладке: его синтаксис предполагает, что после THEN и ELSE должно стоять только по одному оператору. При необходимости поставить несколько операторов их надо заключать в операторные скобки DO-END. Само по себе это неудобно, но не страшно, однако когда программист забывает эти скобки, то начинаются эффекты от неприятных до опасных. Рассмотрим их на простейшем примере.

Пусть надо получить максимальное и минимальное из двух чисел A и B, что дает следующий фрагмент программы:

```
IF A>B THEN DO; MAX=A; MIN=B; END;
       ELSE DO; MAX=B; MIN=A; END;
```

Если программист забыл DO-END после THEN, то он получает синтаксическое сообщение «ELSE без IF», которое приводит начинающего программиста в полное недоумение: как же - вот же IF строчкой выше?! Чтобы научиться правильно понимать диагностические сообщения транслятора, надо научиться смотреть на программу «глазами ЭВМ». Попробуем прочесть фрагмент

```
IF A>B THEN MAX=A; MIN=B; ELSE DO; MAX=B; MIN=A; END;
```

этими глазами. Читаем «IF A>B THEN...». Понятно? Да. Читаем дальше: «IF A>B THEN MAX=A;». Понятно? Да. Читаем дальше: «IF A>B THEN MAX=A; MIN=B;...» Пока все и нам и ЭВМ понятно, но если посмотреть только на этот фрагмент, то становится ясно, что «MIN=B;» - это самостоятельный оператор: раз нет ELSE, то IF закончился. А далее «вдруг» встречается ELSE - ясно, что это ELSE, у которого не было IF. Диагностическое сообщение стало понятным.

Аналогичная ошибка в другом месте: забыты те же DO-END после ELSE - не вызовет синтаксического диагностического сообщения транслятора. С его точки зрения - это синтаксически правильная конструкция, где ветвь ELSE включает только один оператор «MAX=B;», а «MIN=A;» - это самостоятельный оператор, который будет выполняться как при истинном, так и при ложном условии. Эта ситуация гораздо более опасна, чем предыдущая, так как программа успешно транслируется и выполняется, причем в некоторых случаях - при ложном условии - дает правильные результаты. В итоге при не слишком тщательном тестировании ошибку можно и не заметить.

Вложенные ветвления. Вложение неполных ветвлений (т. е. без части ELSE) в другое ветвление также может послужить источником ошибок. Например, фрагмент

```
IF I<N THEN IF A(I)=X THEN K=I;
        ELSE PUT EDIT('ОШИБКА') (A);
```

на первый взгляд при I>N будет печатать слово 'ОШИБКА', но если посмотреть на этот фрагмент с точки зрения ЭВМ, то ELSE относится к ближайшему THEN, которое еще «не закрыто» ELSE.

Таким образом, этот фрагмент будет машиной понят как

```
IF I<N THEN
          IF A(I)=X THEN K=I;
                    ELSE PUT EDIT('ОШИБКА') (A);
       ELSE;
```

а вовсе не как

```
IF I<N THEN DO IF A(I)=X THEN K=I; END;
        ELSE PUT EDIT('ОШИБКА') (A);
```

Если же надо, чтобы получился второй вариант, то приходится или добавить пустую ветвь ELSE, как показано выше, или же не забыть DO-END.

Замечание. *Эти проблемы могли бы не возникать, если бы в ПЛ/1 для IF была своя «закрывающая скобка», как для DO есть «закрывающая скобка» END. Тогда упомянутые выше ошибки просто не могли бы возникнуть, ибо в получающихся конструкциях не возникает никаких двусмысленностей:*

```
IF I<N THEN IF A(I)=X THEN K=I; ENDIF;
        ELSE PUT EDIT ('ОШИБКА') (A); ENDIF;

IF A>B THEN MAX=A; MIN=B; ESE MAX=B; MIN=A; ENDIF;
```

При этом пропуск ENDIF вызывает диагностическое сообщение транслятора.

2.4. Циклы

Общие замечания. ПЛ/1 имеет очень мощную и удобную конструкцию цикла, позволяющую коротко и ясно выразить многие часто встречающиеся ситуации. Однако, как и всегда в подобных случаях, сложная конструкция имеет свои подводные камни, которые иногда дают неожиданные на первый взгляд эффекты.

Циклы делятся на две основные категории: итерационные (DO WHILE(...)), которые не содержат для программиста никаких неожиданностей, и циклы с параметром, которые, собственно, и имеют множество модификаций и множество тонкостей.

Работа цикла с параметром. В общем случае в конструкции цикла

DO «параметр»= «нач. знач» BY «шаг» TO «кон. знач», END;

«нач. знач.», «шаг» и «кон. знач.» могут быть выражениями общего вида, значения которых вычисляются один раз в начале работы цикла и не могут быть изменены в процессе его работы. Рассмотрим примеры, показывающие, как будут работать циклы, в которых делается попытка изменить значение шага и конечного значения.

```
H=1;
DO I=1 BY H TO 10;
  PUT LIST(H); H=-H;
END;
```

Сколько раз и при каких значениях I выполнится тело цикла? Какие значения H будут напечатаны? Вопреки первому впечатлению этот цикл не зациклится, ибо при входе в него будет запомнено, что шаг равен единице, следовательно, цикл выполнится 10 раз со значениями 1=1,2,3,4,5,6,7,8,9,10. Однако печататься будут попеременно значения 1 и -1, т. е. значения H меняться будут, а после завершения цикла H останется со значением +1. Дело в том, что при входе в цикл запоминаются текущие значения выражений «шаг» и «кон. знач.», а не те переменные, через

значения которых они определялись. Таким образом, параметр цикла наращивается на величину, которая была в свое время скопирована из H, но более к H никакого отношения не имеет, что позволяет меняться H, не меняя значение шага.

То же самое относится и к циклу

```
N=10;
DO I=1 TO N;
... N=N-1;
END;
```

Цикл выполнится 10 раз со значениями 1=1,2,3,4,5,6,7,8,9,10, при этом после цикла значение N станет нулевым. Фактически при трансляции такого цикла транслятор вводит две вспомогательные переменные ВСПОМ1, ВСПОМ2, у которых нет известных программисту идентификаторов, и реализует цикл так:

```
ВСПОМ1=«шаг»; ВСПОМ2= «кон. знач.»; «параметр»=«нач. знач.»;
DO WHILE((ВСПОМ2- «параметр»)*ВСПОМ1 >=0 );
  «тело цикла»;
  «параметр»= «параметр»+ВСПОМ1;
END;
```

Из этой записи становится ясно, что в процессе работы цикла переменные ВСПОМ1 и ВСПОМ2 не меняются, при том что переменные, входящие в «шаг» или «кон. знач.», могут меняться. Случаи, когда может потребоваться цикл с меняющимся шагом или изменением в процессе работы цикла конечного значения, в практике встретиться могут, тогда приходится использовать WHILE-цикл. Например, вместо цикла

```
DO X=A BY H TO B;
... H=H+DH;
END;
```

придется использовать цикл

```
X=A;
DO H=H BY DH WHILE(X<=B);
...X=X+H;
END;
```

Отметим, что здесь параметром цикла стал шаг. Запись вида «DO H=H» допустима и означает «меняя H от текущего значения с шагом ...».

Принудительное изменение параметра цикла. Как видно из записи цикла, преобразованной к форме DO WHILE, параметр цикла не защищен от доступа другими операторами программы. И действительно, есть возможность изменять параметр цикла внутри тел цикла, хотя делать это не рекомендуется во избежание ошибок. рассмотрим, какие получаются при этом эффекты. В цикле

```
DO I=1 TO N;
... I=I+1;
```

```
END;
```

параметр цикла I на каждом шаге будет меняться дважды: в теле цикла его увеличивает оператор присваивания, а кроме того, его увеличивает заголовок цикла. В итоге цикл будет выполняться 5 раз при 1=1,3,5,7,9; при выходе из цикла I будет иметь значение 11. Если вместо «I=I+1» в теле цикла поставить оператор «I=I-1», то получим зацикливание: этот оператор уменьшает значение I, затем заголовок цикла увеличивает его, в итоге значение I остается прежним.

Выводы таковы: хотя нет формального запрета менять параметр цикла в теле цикла, но делать этого не рекомендуется. Иногда таким приемом пользуются для принудительного досрочного выхода из цикла:

```
DO I=1 TO N;
   IF ... THEN I=N; ELSE ...
END;
```

но лучше для этих целей использовать GOTO или специальный оператор LEAVE, так как это проще понимается, а потому менее подвержено ошибкам.

Цикл с действительным параметром. В отличие от ряда языков, где допускается цикл только с целым параметром, ПЛ/1 позволяет организовать цикл и с действительным параметром, что дает возможность удобно записывать циклы вида

```
DO X=A BY H TO B;
```

которые часто встречаются в разных численных расчетах (вычисление интегралов, например).

Понятно, что при A=1, B=2 и H=0.1 такой цикл должен выполняться 11 раз. Однако надо отметить, что поскольку X - переменная действительная, а действительные значения приближенные, то после десятикратного добавления к X значения H накапливается погрешность и при A=1, B=2, H=0.1 последнее значение X может оказаться как чуть меньше 2, так и чуть больше 2. В первом случае ничего особенного не произойдет, во втором - при последнем значении X тело цикла уже не будет работать, так как X выйдет за пределы B. Именно из-за таких особенностей работы с действительными числами и запрещены в ряде языков циклы с действительными параметрами, потому что трудно сказать, сколько раз будет выполняться тело цикла.

Простейший способ избавиться от этого недостатка - это записать заголовок цикла в виде `DO X=A BY H TO B+H/2;` учтя погрешности действительных переменных. В большинстве случаев этого достаточно, чтобы точно определить число повторений цикла, однако есть ситуации (они описаны в гл. 5), когда с погрешностями машинной арифметики приходится бороться более сложными методами.

Комбинированные циклы. Очень удобной конструкцией ПЛ/1, почти не имеющей аналогов в других языках, являются циклы, имеющие в заголовке как параметр, так и условия WHILE и/или REPEATE:

DO «парам.»= «нач. зн.» [BY «шаг»} [TO «кон. зн.»} [WHILE («усл1»)] [REPEAT («выражение»)];

Наряду с удобством эти циклы имеют ряд особенностей и подводных камней.

Во-первых, заметим, что любая взятая в квадратные скобки часть может отсутствовать. Если отсутствует часть WHILE, то не проверяются соответствующие условия; если отсутствует часть TO, то не проверяется выход параметра за диапазон «нач. зн.»... «кон. зн.», что естественно. Если не указан шаг (отсутствует часть BY), то по умолчанию он предполагается единичным, однако если не указаны ни шаг, ни конечное значение, то цикл выполняется не более одного раза. Таким образом, циклы

```
DO I=1 BY 1 TO M;  и
DO I=1 TO M;
```

работают одинаково, а циклы

```
DO I=1 BY 1 WHILE(I<=N);  и
DO I=1 WHILE (I<=N);
```

работают по-разному: во втором варианте тело цикла отработает не более одного раза, ибо наращивания параметра цикла на единицу происходить не будет.

Очень важной деталью комбинированных циклов является то, *что сначала проверяется выход параметра за диапазон, а затем только, если параметр за диапазон не вышел, проверяется условие, стоящее после WHILE*. Это позволяет удобно писать циклы поиска в массивах элементов, удовлетворяющих каким-то условиям. Характерной особенностью циклов поиска являются два случая выхода из цикла: элемент найден и цикл завершается досрочно; элемента нет и цикл завершается по исчерпанию элементов массива. Например, поиск в массиве A размера N элемента, равного X, записывается

```
DO I=1 BY 1 TO N WHILE(A(I)^=X); END;
```

Этот цикл завершает работу либо по исчерпании элементов массива, когда I>N, либо по невыполнению условия после WHILE, т. е. элемент найден. Существенной особенностью циклов ПЛ/1 является то, что после завершения цикла параметр цикла сохраняет последнее полученное значение, т. е. то значение, которое уже вышло за диапазон. Проверив значение параметра после окончания цикла, можно узнать, по какой причине завершился цикл, и как-то обработать либо найденный элемент массива, либо ситуацию отсутствия элемента (например, напечатать предупреждающее сообщение):

```
DO I=1 BY 1 TO N WHILE(A(I)^=X); END;
IF I>N THEN <обработка отсутствия элемента>;
       ELSE <обработка элемента A(I)>;
```

Отметим, что проверка

```
IF A(I)^=X THEN <обработка отсутствия элемента>;
           ELSE <обработка элемента A(I)>;
```

недопустима, ибо при отсутствии элемента цикл завершится со значением I равным N+1, и попытка сравнить элемент A(N+1) с X приведет к ситуации SUBSCRIPTRANGE (ERROR(16) для компилятора PL/1-KT).

В таком использовании оператора цикла весьма существенно то, что сначала проверяется параметр, а затем условие. Но для поиска элемента массива нельзя использовать цикл

```
DO I=1 BY 1 WHILE(I<=N & A(I)^=X); END;
```

так как при отсутствии элемента он может дать неожиданные эффекты. Если I>N, то независимо от результата сравнения A(1)^=X значение условия ложно, однако, даже убедившись в том, что I>N, многие ЭВМ все же попытаются вычислить значение выражения A(I)^=X, т. е. обработать (N+1)-й элемент массива из N элементов, что даст неопределенный эффект. В трансляторе PL/1-KT есть ключ трансляции «W», позволяющий не допускать вычисления последующих выражений в условии.

Если не включен особый режим контроля индексов (ситуация SUBSCRIPTRANGE), то попытка «залезть за массив» останется незамеченной и из какой-то ячейки памяти, находящейся рядом с ячейками, отведенными под массив, будет взято какое-то значение. Это лучший вариант, ибо результат сравнения не важен. Если же контроль индексов включен, то будет выдано сообщение «выход индекса за границу массива» (ERROR(16) в PL/1-KT) и выполнение программы будет прервано. Наконец, в некоторых случаях, которые рассмотрены в гл. 1, могут быть выданы «странные» сообщения «защита памяти» или еще что-нибудь подобное.

2.5. Оператор перехода и метки

Использование операторов перехода. ПЛ/1 имеет полный набор структурных конструкций, а значит, на нем можно писать программы, вообще не используя оператор GO TO - есть строго математический способ доказательства того, что любую программу по чисто формальному алгоритму, не вникая в подробности того, что она делает, можно преобразовать в программу без GO TO. Вопрос заключается только в том, стоит ли это делать и стоит ли избегать использования GO TO в принципе.

Идея известного теоретика программирования Е. Дейкстры писать программы без GOTO была связана с тем, что беспорядочное использование GO TO затрудняет понимание программы и вносит ошибки. Но применение GO TO в малых дозах в

отдельных случаях может облегчить понимание программы, а значит, как следствие, снизить вероятность появления ошибок.

Можно сформулировать три случая, в которых целесообразно использование GO TO:

- ситуации поиска, в которых требуется досрочный выход из цикла, причем выход должен быть выполнен из середины тела цикла, в этом случае проще использовать GO TO (или оператор LEAVE, который представляет собой неявный GO TO), чем фокусы с дополнительными проверками, нужно ли выполнять оставшуюся часть тела;
- ситуации, связанные с обработкой прерываний: например, не каждый программист сообразит, как преобразовать фрагмент

  ```
  ON ZERODIVIDE GO TO M;
  ... A=B/C+D/E+F/H; ...
  ```

  к виду без GO TO и вряд ли кто согласится признать, что результат более понятен, чем исходный вариант;
- ситуации, когда необходимо построить нестандартную управляющую конструкцию (пример приведен ниже).

Примером простейшей нестандартной управляющей конструкции является ситуация, когда в ветвлении надо сделать две проверки в условии, причем вторую можно делать тогда, когда дала положительный ответ первая:

```
IF I<N THEN
                IF A(I)=X THEN <обработка элемента A(I)>;
,                               ELSE <обработка отсутствия элемента>
        ELSE <обработка отсутствия элемента>;
```

Понятно, что по причинам, описанным при рассмотрении комбинированных циклов, это нельзя записать в виде

```
IF I<N & A(I)=X THEN <обработка элемента A(I)>;
                ELSE <обработка отсутствия элементов>;
```

и простейший способ избежать повторения одинаковых частей дважды - это применить GO TO:

```
IF I<N THEN GO TO M;
        ELSE IF A(I)=X
                THEN  M:<обработка элемента A(I)>
                ELSE   <обработка отсутствия элемента>;
```

Нетрудно заметить, что этот вариант и проще, и понятнее, и эффективнее (это легко проверить по табл. 1.1) любого другого варианта. Такое программирование нарушает локальную структурность программы, но вся эта конструкция в целом - ветвление с двумя проверками и двумя ветвями - имеет один вход и один выход, как и положено структурным конструкциям.

Другое применение GO TO - это моделирование отсутствующих в языке конструкций. Например, в PL/1-KT нет цикла UNTIL, который нужен во многих итерационных вычислительных методах, когда надо сначала сделать шаг вычислений, а затем выполнять проверку, нужно ли завершать цикл. Такой цикл гораздо проще реализовать с помощью GO TO, чем выполнять обходные маневры.

Например, итерационный метод приближенного вычисления квадратного корня требует UNTIL-цикла

```
Y=X/2;
DO UNTIL(ABS(Y-Y1)<EPS);
  Y1=Y; Y=(Y+X/Y)/2;
END;
```

Всевозможные способы обойтись в ПЛ/1 без GO TO с помощью WHILE-цикла путем выполнения одного шага вычислений до цикла

```
Y1=X/2; Y=(Y+X/Y)/2;
DO WHILE(ABS(Y-Y1)>EPS);
  Y1=Y; Y=(Y+X/Y)/2;
END;
```

или путем различных способов «обмана» проверки условия, чтобы на первом шаге «проникнуть» в тело цикла

```
Y=X/2; Y1=Y+EPS;
DO WHILE (ABS(Y-Y1)>=EPS);
  Y1=Y; Y=(Y+X/Y)/2;
END;
```

уступают и по эффективности (см. табл. 1.1), и по понятности (а значит, и по ошибкоустойчивости) простейшему варианту с GO TO:

```
Y=X/2;
C: Y1=Y; Y=(Y+X/Y)/2;
IF ABS(Y-Y1)>=EPS THEN GOTO C;
```

Вывод таков: избегание оператора GO TO любой ценой нецелесообразно, его надо применять там, где без него хуже, чем с ним.

Переменные типа метки. Метки так же, как и другие значения, можно запоминать в переменных специального типа, которые описываются как DCL <идентификатор> [<размеры массива>] LABEL [<другие атрибуты>];

Значение такая переменная может получить оператором присваивания, что здесь не рассматривается, так как практически не имеет полезных применений, либо присваиванием начальных значений.

Сомнительное утверждение: присваивание значений метки-переменной других значений широко и очень удобно используется в некоторых задачах, например при программировании конечных автоматов.

Использована такая переменная может быть только в операторе перехода: переход выполняется на ту метку, которая была присвоена этой переменной.

С современных позиций использование средств, которые изменяют структуру программы в процессе ее выполнения, не приветствуется, так как чревато ошибками, поэтому присваивание меточным переменным выполнять не рекомендуется.

Остается использовать меточные переменные со значениями, полученными начальными присваиваниями. Практически единственное полезное применение этой конструкции - так называемый переключатель:

```
DCL P(1:4) LABEL STATIC INIT(A,B5,CC,K); ...
GOTO P(I)
A:  <оператор 1> …
B5: <оператор 2> …
CC: <оператор 3> …
K:   <оператор 4>  …
```

Здесь описан, массив меточных переменных P, состоящий из четырех элементов, которым начальными присваиваниями присвоены метки A, B5, CC и K соответственно (упомянутые после INIT метки обязательно должны где-то встречаться в программе). Где-то в теле программы стоит оператор GO TO P(I); если I=1, то он выполнит переход на метку, записанную в P(I), то есть на A; если I=2, то - на B5 и т. д. Иными словами фактически получается более короткая и удобная запись разветвления на несколько направлений, которую можно записать и без GO TO, хотя и ценой потери краткости и скорости программы:

```
IF          I=1 THEN  DO; <оператор 1>;… END;
ELSE IF     I=2 THEN  DO; <оператор 2>;… END;
ELSE IF     I=3 THEN  DO; <оператор 3>;… END;
ELSE                  DO; <оператор 4>;… END;
```

2.6. Оператор останова

Назначение. Оператор останова предназначен для прекращения работы программы, однако программа завершает свою работу и без него, дойдя до END, соответствующего PROCEDURE головной программы. Поэтому оператор STOP используют для аварийного останова в случае обнаружения какой-либо ошибки в данных, которая не позволяет продолжать работу, или если алгоритм построен так, что останов удобнее поставить не в конце алгоритма. Примером второй ситуации является алгоритм, работающий в цикле «ввод - обработка - вывод результатов», который завершает свою работу при вводе какого-то специального значения:

```
M: GET LIST(<данные>);
IF <проверка на конец работы> THEN STOP;
<обработка>
<вывод результатов>
GO TO M;
```

Действие. Согласно документации ПЛ/1 оператор STOP считается оператором аварийного останова и в упомянутом выше случае применяться не должен. Здесь, однако, имеются некоторые противоречия. Оператор STOP вызывает ситуацию

FINISH, а не ERROR, т. е. влечет нормальное, а не аварийное завершение программы. Но в то же время программа по завершении вырабатывает так называемый код возврата, который на работу самой программы не влияет, но сигнализирует операционной системе, как закончилась работа: нормально или аварийно. Завершение программы по END головной программы дает неопределенный код завершения, а завершение по STOP может дать заданный код завершения при выполнении оператора STOP(X), где X имеет тип BIN FIXED(7). Если программа выполняется сама по себе, то это ни на что не влияет, если же программа выполняется по вызову из некоторой многошаговой процедуры выполнения задания и ее результатами пользуется следующая программа, то процедура может включать проверку успешности выполнения этой программы - и тогда неопределенный код завершения будет воспринят этой проверкой как аварийное завершение.

Глава 3. ВВОД-ВЫВОД

3.1. Общий порядок работы

Взаимодействие списка ввода-вывода и формата. Общее правило взаимодействия списка ввода-вывода и формата просто: формат перебирается в цикле, список ввода-вывода - один раз. По исчерпании списка работа оператора ввода-вывода завершается. Однако на практике эти правила создают некоторые трудности. Пусть имеется пара операторов вывода

```
PUT EDIT ((A(I) DO I=1 TO 20)) (10 F(7,3), SKIP);
PUT EDIT ('КОНЕЦ') (A);
```

По идее первый оператор выводит данные по 10 значений в строке и делает переход на следующую строку, т. е. должны получиться две строки по 10 чисел и третья строка со словом «КОНЕЦ». Но вопреки первому впечатлению на выводе будет получено только 2 строки: первая - из 10 чисел, вторая - из 10 чисел и слова «КОНЕЦ».

Чтобы понять эффект, проследим работу первого оператора. После вывода первых 10 значений список ввода-вывода не исчерпан, поэтому продолжается просмотр формата, выполняется переход на новую строку в соответствии с элементом SKIP, затем формат просматривается с начала. После вывода следующих 10 элементов список ввода-вывода исчерпан, следовательно, оператор работу завершает, а SKIP при этом не отработал, т. е. переход на новую строку не выполняется. В подобных ситуациях принудительный переход на новую строку следует вынести в начала форматов:

```
PUT EDIT ((A(I) DO I=1 TO 20))(SKIP,10 F(7,3));
PUT EDIT ('КОНЕЦ') (SKIP,A);
```

или же оформить первый оператор так:

```
PUT EDIT ((A(I) DO I=1 TO 20),' ')(10 F(7,3),SKIP,A);
```

или так:

```
PUT EDIT ((A(I) DO I=1 TO 20))(SKIP,10 F(7,3)); PUT SKIP;
```

Несколько большие проблемы этот эффект создает при форматном вводе, так как пара операторов

```
GET EDIT ((A(I) DO I=1 TO 20)) (10 F(7,3),SKIP);
GET EDIT (S)(A(10));
```

приведет к вводу строки A со второй строки исходных данных, что может быть нежелательно (естественнее начинать новые по смыслу данные с новой строки), а перенос SKIP в начало:

```
GET EDIT ((A(I) DO I=1 TO 20))(SKIP,10 F(7,3));
GET EDIT (S)(SKIP,A(10));
```

приведет к тому, что первый оператор GET начнет с пропуска строки (первая строка исходных данных будет пропущена, т. е. при наборе надо пропустить строку перед данными). Видимо, наилучший вариант, который не требует пустых строк перед данными, это вариант

```
GET EDIT ((A(I) DO I=1 TO 20))(COLUMN(1),10 F(7,3));
GET EDIT (S)(COLUMN(1),A(10));
```

Ситуация «конец файла». Ситуация ENDFILE возникает при обнаружении конца файла при чтении данных, по оператору

ON ENDFILE(<файл>..) ...; программист может обработать возникновение конца файла так, как ему надо.

В PL/1-KT эта ситуация может быть и для выходных файлов, если на диске не хватает места для записи.

Преобразования при выводе. Автоматические преобразования при выводе иногда дают на первый взгляд не совсем понятные эффекты. Они связаны в основном с выдачей по некорректному формату и возникают при ошибках в организации взаимодействия списка вывода и формата. Пусть, например, имеются числовой массив, описанный

```
DCL A(5) FLOAT STATIC INIT(1,2,3,4,5);
```

и оператор вывода

```
PUT EDIT(':', A, ' :' ) (A(4),5 F(3),A);
```

обеспечивающий вывод в виде строки

```
:   1  2  3.  4 5  :
```

По ошибке этот оператор записали в виде

```
PUT EDIT(':',A,' :') (A(4),F(3),A);
```

На печати получим следующий результат:

.:....1.2.00000E-00.3.0..4.5.00000E+00.: при этом никаких диагностических сообщений не последует. Дело в том, что массив представляет для вывода пять значений. Первое выводится по формату F(3) и дает ожидаемый результат « 1», второе - по формату A, третье - по формату A(4), так как перебор формата повторяется с начала, четвертое - опять по «своему» формату F(3), пятое - опять по формату A. И хотя очевидно, что вывод числового значения по строковому формату - это скорее всего ошибка, но ПЛ/1 просто преобразует значение в строку по правилам преобразования и напечатает их. Поскольку атрибуты массива - FLOAT(6), где точность (6) предполагается по умолчанию, то число преобразуется в строку с шестью значащими цифрами и с порядком. При выводе по формату A печатаются

все символы строки, при выводе по формату A(4) печатаются только первые четыре цифры.

Особенность работы при зацикливании. При работе программы вывод на печать или в файл на диске, который затем распечатывается, происходит не в момент выполнения операторов PUT. Этот оператор приводит только к занесению символов в так называемый буфер, который может хранить несколько символов (обычно одну строку 80 символов). Следующий PUT пополняет буфер очередной порцией символов и т. д. Собственно вывод (непосредственно на выводное устройство: на экран или АЦПУ или в промежуточный файл) происходит либо, когда в буфере оказалась полная строка в 80 символов, либо когда обрабатывается элемент формата SKIP, т.е. когда становится ясно, что в этой строке более ничего напечатано не будет. По завершении программы, т.е. по ситуациям типа ERROR, неполный буфер, содержащий последнюю информацию, выводится в файл или на устройство.

Если же программа зациклилась и снята по команде самого программиста, то последняя строка так и остается в буфере и оказывается не напечатанной. Пусть, например, имеется фрагмент программы:

```
PUT LIST(A,B);
…
PUT LIST(C);
…
PUT SKIP LIST('ЗНАЧЕНИЕ');
…
PUT EDIT(K)(F(3));
…
PUT EDIT(L)(F(3));
…
PUT EDIT (M)(SKIP, F(3));
…
PUT EDIT(N) (F(3));
…
PUT EDIT(P,Q,R) (F(3));
```

Первый оператор PUT LIST выводит в буфер значения A и B. По правилам вывода по LIST буфер будет заполнен до 48-й позиции, следующий PUT LIST дополняет буфер очередным значением (с 49-й позиции в буфер заносятся символы, представляющие значение переменной C, но вывода на внешние устройства пока нет). Следующий оператор PUT SKIP LIST начинает с отработки режима SKIP: содержимое буфера выводится на внешнее устройство и буфер очищается, а затем в него заносятся символы «ЗНАЧЕНИЕ». Далее буфер пополняется значениями K и L

и выводится при выполнении оператора PUT EDIT (M)(SKIP, F(3)); при обработке элемента формата SKIP. Теперь предположим, что между операторами PUT EDIT(N)… и PUT EDIT(P,Q,R)... программа зациклилась. Тогда программист не обнаружит на выдаче значений M и N - они «зависли на буфере». Не зная этой особенности ПЛ/ 1, программист начинает искать зацикливание между операторами PUT EDIT(L)... и PUT EDIT(M)... и, естественно, его там не находит.

Еще более шокирующей является ситуация зацикливания после оператора печати вида

```
PUT EDIT((A(I) DO I=1 TO 50)) (SKIP, 10 F(5,1));
```

ибо на печати оказываются четыре строки, а не пять, и программист ищет зацикливание в операторе печати, которое, разумеется, найти не может.

Рекомендации в подобных ситуациях таковы. Во-первых, искать зацикливание следует в более широком диапазоне - от той печати, которая дала последнюю строку на выдаче, и до следующей печати с режимом SKIP. Если же зацикливание локализовать не удается, то после каждой печати следует добавить оператор PUT SKIP, который принудительно выводит буфер на внешнее устройство.

Понятно, что вывод займет много строк, но как временная мера это поможет определить место зацикливания.

3.2. Синтаксическая правильность операторов

Анализ правильности списка ввода-вывода. Список ввода-вывода является конструкцией, довольно чувствительной к ошибкам: в нем не допускаются лишние скобки, а пропуск скобок иногда приводит к непонятным (на первый взгляд) диагностическим сообщениям. Наконец, при некоторых ошибках ЭВМ будет делать не то, что имелось в виду. В итоге организация более или менее сложного ввода-вывода доставляет начинающему программисту много хлопот, поэтому надо хорошо владеть техникой проверки операторов ввода-вывода. В процессе этой проверки приходится искать ответы на три вопроса:

1. Правильно ли то, что записано, с точки зрения ЭВМ?
2. Если правильно, то что ЭВМ сделает?
3. Соответствует ли то, что она сделает, тому, что надо нам?

Рассмотрим ввод двух массивов из 10 элементов, организованный так, что элементы массивов чередуются: A(1), B(1), A(2), B(2), и т. д. Правильный оператор ввода имеет вид:

```
GET LIST((A(K), B(K) DO K=1 TO 10));
```

Первая типовая ошибка - пропуск одной пары скобок:

```
GET LIST(A(K), B(K) DO K=1 TO 10);
```

С точки зрения ЭВМ получается список ввода, состоящий из двух элементов (они выделены жирным), первый из которых есть переменная с индексом, а второй не

подходит ни под одну из дозволенных конструкций (как циклический он не воспринимается, так как нет окружающих циклический элемент скобок). Результат: сообщение «ошибка в списке ввода-вывода».

Вторая типовая ошибка - лишние скобки вокруг А(К) и В(К):

```
GET LIST(((A(K). B(K)) DO K=1 TO 10)):
```

С точки зрения ЭВМ список ввода состоит из одного элемента, который является циклическим, так как заключен в скобки . Циклический элемент, в свою очередь, содержит список элементов, состоящий из одного элемента. Так как этот элемент заключен в скобки, то ЭВМ считает его циклическим, внутри него обнаруживает список из двух элементов, а заголовка цикла нет. Результат: сообщение, как и ранее.

Третья типовая ошибка - запятая перед заголовком цикла:

```
GET LIST((A(K), B(K), DO K=1 TO 10));
```

С точки зрения ЭВМ циклический элемент содержит список, состоящий из трех элементов: элементы разделяются запятыми, значит, две запятые разделяют три элемента, третий элемент - это описанная по умолчанию переменная с идентификатором DO. В ПЛ/1 служебные слова не резервированы и распознаются как служебные только в определенном контексте, в программе можно использовать идентификаторы, совпадающие по написанию со служебными словами. Результат: сообщение «ошибка в заголовке цикла» или «неопределенная синтаксическая ошибка», относящаяся к непонятой части оператора.

Еще одна ошибка - заголовок цикла не последняя справа часть:

```
GET LIST((A(K,J) DO K=1 TO 10, B(K))):
```

К сожалению, многие трансляторы ПЛ/1 не указывают то место в операторе, к которому относится диагностическое сообщение. И автору приходилось многократно наблюдать «битвы» программистов с операторами ввода-вывода, в процессе которых они расставляли знаки препинания разными способами,, пытаясь добиться хотя бы синтаксической правильности оператора. Однако косвенный намек на место ошибки трансляторы все же дают в виде сообщений вида *«элемент '.....' является ошибочным и пропускается»*. Как правило, таких сообщений на один оператор следует множество - обращать внимание надо на первое из них; в нем в кавычках вместо точек указан тот элемент, с которого транслятор прекратил понимание оператора.

Второй пример - ввод массива С(1:5) и матрицы А (1:5,1:10) так, что вводится значение С(1), затем вся первая строка матрицы, значение С(2) и вторая строка матрицы и т. д. Такой ввод реализуется оператором

```
GET LIST((C(1),(A(1,K) DO K=1 TO 10) DO I=1 TO 5));
```

Предположим, что в результате ошибки оказалась неправильно поставлена третья скобка - она оказалась перед С(1):

```
GET LIST(((C(1),A(1,K) DO K=1 TO 10) DO I=1 TO 5)):
```

С точки зрения ЭВМ это абсолютно правильный оператор, что трудно проверить описанным выше методом, содержащий в списке ввода один циклический элемент, который, в свою очередь, также содержит один циклический элемент. В результате при первой отработке внутреннего цикла первое значение попадет в С(1), второе - в А (1,1), как и хотел программист; затем при К=2 третье значение, предназначенное для А(1,2), попадет опять в С(1), стерев предыдущее значение, а затем значение, предназначенное для А(1,3), попадет в А(1,2) и т. д. В итоге будет сделана попытка ввести 100 значений вместо 55. Если в данных этих значений просто нет, то получим сообщение «конец файла» (ситуация ENDFILE). Если за подготовленными 55-ю значениями для данного оператора ввода идут какие-то значения другого типа, получим сообщение «незаконное преобразование» (ситуация CONVERSION). Наконец, если в данных имеется еще 45 значений нужного типа, предназначенных для последующего ввода, возможно, другим оператором ввода, то этот оператор (худший для программиста вариант) благополучно завершит работу, введя некоторые странные значения, а упомянутые диагностические сообщения появятся при исполнении какого-то следующего оператора ввода, возможно, после длительной работы ЭВМ по неверным данным.

Подобные же ситуации возможны при выводе. Например, оператор вывода с таким же списком ввода-вывода, как в последнем примере ошибки, просто выведет 100 значений, в которых будет сложно разобраться, и программа продолжит работу. Алгоритм анализа списка ввода-вывода основан на том, что в списке выделяются элементы и анализируются по отдельности.

Причем элементы выделяются по формальному признаку: от запятой до запятой. Каждый элемент проверяется по списку допустимых конструкций. Если же элемент циклический, то после проверки заголовка цикла выделяется внутренний список этого циклического элемента и к нему применяется та же процедура.

Составление операторов ввода-вывода. Пусть даны три массива: целые массивы А(1:10) и В(1:10) и действительный массив С(1:10).

Для определенности положим, что они содержат значения, задаваемые формулами: A(I)=2*I; B(I)=5*I; C(I)=10*I+0.5:

Рассмотрим следующие вспомогательные задачи. Задачи на составление только списка ввода-вывода:

а) написать список ввода-вывода, выводящий весь массив А, затем весь массив В, затем весь массив С;

б) написать список ввода-вывода, выводящий элементы массивов в таком порядке: А(1), В(1), С(1), А(2), В(2), С(2) и т. д.

Задачи на составление только формата:

1) написать формат, обеспечивающий печать двух строк из 10 целых чисел каждая и строки из 10 действительных чисел,

2) написать формат, обеспечивающий печать десяти пар строк, где первая строка каждой пары содержит два целых числа, а вторая - одно действительное.

Список ввода-вывода определяет, *что* будет печататься. В соответствии с условиями задач а и б нужные списки имеют вид:

а) (A,B,C)

б) ((A(I),B(I),C(I) DO I=1 TO 10))

Формат определяет, *как* будет печататься. Строго говоря, задачи 1 и 2 неразрешимы, ибо формат не определяет количество строк: он может повторяться несколько раз, поэтому формат определяет бесконечное количество повторений одинаковых групп полей. Форматы

1) (2(SKIP.10 F(5)), SKIP,10 F(5,2))

2) (SKIP,2 F(5), SKIP,X(3),F(5,2))

дают нам нужное решение с той поправкой, что формат 1 задает бесконечное повторение троек строк 10 раз а формат 2 - бесконечное повторение пар строк. Теперь объединим что и как. Каждый из списков можно объединить с любым из форматов. Расставив идентификаторы переменных в соответствующих полях, мы получим четыре варианта шаблонов.

Вариант а1 дает вполне естественную печать трех массивов в три строки, а вариант б2 - вполне естественную печать в столбец.

Варианты б1 и а2 скорее всего могут возникнуть из-за ошибки, однако посмотрим, что ЭВМ будет делать.

```
    2     5    11     4    10    21     6    15    31     8
   20    41    10    25    51    12    30    61    14    35
70.50 16.00 40.00 80.50 18.00 45.00 90.00 20.00 50.00 00.50
```

Внешне как будто бы все в порядке: получили, как и хотели, три строки - две из целых чисел, одну из действительных, только значения в них какие-то странные (на первый взгляд). Анализ подобной выдачи требует внимательного прослеживания того, что и как выводилось, в соответствии с правилами взаимодействия списка ввода-вывода и формата. А это удобнее всего сделать, отдельно выписав, *что* выводится, и отдельно - *как* выводится, собрать *что и как* в шаблон и наложить получившийся шаблон на имеющуюся выдачу.

Глава 4. ОСОБЕННОСТИ РАБОТЫ СО СТРОКАМИ

4.1. Ввод-вывод строк

Перебор списка данных и формата. Собственно ввод-вывод по форматам A и B никаких проблем не ставит, однако при вводе-выводе строк надо четко представлять, на что идет один элемент формата. Пусть даны строковые переменные с описаниями

```
DCL C CHAR(100), CP CHAR(100) VAR, M(100) CHAR(1);
```

Строка C, как и строка CP - это элементарное данное, а M - это массив из 100 односимвольных строк. Ввод этих переменных выполняется операторами

```
GET EDIT(C)(A(100));
GET EDIT(CP)(A(100));
GET EDIT(M) (100 A(1));
```

Типовая ошибка - применение для ввода этих строк операторов

```
GET EDIT(C) (100 A(1));
GET EDIT(CP)(100 A(1));
GET EDIT(M) (A(100));
```

Здесь предполагается, что раз надо ввести 100 символов, то надо записать ввод 100 раз по одному символу, как это сделано в первых двух операторах, или же, если надо ввести всего 100 символов, то можно написать A(100), как в третьем операторе. Синтаксически эти операторы правильны, т. е. диагностических сообщений ЭВМ не даст и при выполнении их тоже не будет обнаружено ошибок. В первых двух операторах в списке ввода-вывода имеется один элемент, поэтому из формата также будет взят один элемент A(I), из вводной строки будет считан один символ. При записи в строку C этот символ по правилам присваивания дополнится 99 пробелами, на этом ввод завершится, неиспользованные 99 элементов формата машине не мешают. При присваивании введенного значения строке CP дополнения пробелами не произойдет, а все остальное полностью аналогично.

В третьем операторе в списке ввода-вывода задан один идентификатор, но это идентификатор массива, а потому он представляет 100 элементов. В формате имеется один элемент, значит, формат будет просмотрен 100 раз. В соответствии с форматом из вводной строки считается 100 символов, что даст 100-символьную строку, и эта строка будет записана в элемент M(1): по правилам присваивания 99 правых символов будут отсечены, затем будут считаны еще 100 символов, из которых только первый попадет в M(2), и т.д. В этом случае ЭВМ может выдать сообщение об ошибке только, в случае, когда общее число вводимых данных меньше 10000 символов. Если же оно больше, то ввод пройдет благополучно, далее будет выполнена работа с введенными значениями, а какие-либо ошибки могут возникнуть только при работе следующего оператора ввода, когда ему не хватит данных или данные окажутся неподходящего типа.

4.2. Особенности работы со строками постоянной длины

Общие замечания. При работе со строками постоянной длины проблемы возникают из-за того, что эти строки всегда содержат столько символов, сколько задано в их описании. При попытке присвоить им более короткое значение оно автоматически, без предупреждения дополняется пробелами (или нулями для битовых строк), и эти дополнительные символы участвуют в операциях, создавая иногда не сразу понятные эффекты.

Сцепление. Пусть имеются три строки с описаниями

```
DCL A CHAR(4), B CHAR(6), C CHAR(5);
```

и выполняются следующие присваивания:

```
A='ПР'; B='СТУ';   A=A!!B; C=A!!B;
```

Вопреки первому впечатлению, после этих присваиваний значением переменной A будет строка 'ПР__', а значением C - строка 'ПР_C' (символы подчеркивания использованы вместо пробелов, чтобы было ясно, сколько пробелов имеется в значении). Дело в том, что после первых двух присваиваний переменные A и B получат дополненные пробелами значения 'ПР' и 'СТУ' соответственно. При выполнении операции A!!B получится 10-символьное значение 'ПР__СТУ___', а при засылке этого значения в A последние шесть символов будут отсечены. Аналогично объясняется и результат последнего присваивания. Такие эффекты получаются и при работе с битовыми строками.

Можно, разумеется, найти задачи, где автоматически дополнение-отсечение строк оказывается полезным, однако они редки. Поэтому пользоваться строками постоянной длины следует только тогда, когда действительно в строке будут значения с постоянным числом символов или битов.

Поиск подстроки. Для тех же строк A, B, C последовательность операторов

```
B='ABC'; A='A'; K=INDEX(B,A);
```

даст значение K=0, поскольку в строке B будет искаться не буква «A», а подстрока 'A_'. Это еще раз подтверждает данную выше рекомендацию по использованию строк постоянной длины.

Присваивание подстроке. При присваивании значения подстроке дополнение-отсечение происходит только в пределах подстроки (подстрока в присваивании всегда работает как строка с постоянной длиной, заданной в SUBSTR), остальные символы не затрагиваются. Если в строке из 10 символов присвоили значение только подстроке из одного символа, то остальные девять заполнены пробелами не будут.

Например, пусть имеется строка с описанием

```
DCL S CHAR(10);
```

для которой выполнено присваивание

```
SUBSTR(S,2,3)='AB';
```

а до него другой работы с этой строкой не было. Значение 'AB' будет дополнено пробелом до длины подстроки, равной 3. В итоге получат значения только 2-й, 3-й и 4-й символы, а прочие останутся неопределенными. В итоге строка получит значение '?AB ??????', где знак вопроса показывает неопределенное значение символа.

Атрибут VAR подстроки. Отметим, что согласно документации ПЛ/1 подстрока всегда имеет атрибут VAR, хотя ведет себя как .строка *постоянной* длины. Это означает, что присваиванием подстроке более короткого или более длинного значения нельзя изменить длину, значения всей строки: присваиваемое значение будет дополняться пробелами или усекаться до длины подстроки.

Атрибут VAR. означает в данном случае только то, что такой атрибут будут получать все значения, в которых участвовала эта подстрока. Например, значение `ST !! SUBSTR(S,I,L)` всегда будет иметь атрибут VAR, независимо от описаний строк S и ST1. Влияние этого атрибута может сказаться только при передаче такого выражения в качестве параметра подпрограммы, если формальный параметр не имеет атрибута VAR (впрочем, это не влечет за собой неприятных последствий).

4.3. Строки без значений

Строки переменной длины. В ПЛ/1 строки переменной длины до первого присваивания содержат нуль символов, и использование строки с не присвоенным значением никаких особенных ошибок повлечь не может.

Строки постоянной длины. В отличие от строк переменной длины, которые до первого присваивания не содержат символов, строки постоянной длины всегда содержат какие-то символы (или биты, если строка битовая). Если напечатать строку (подстроку), то те позиции в ней, которые не получили значения (и имеют нулевые байты), распечатываются как пробелы и на экране выглядят как пробелы, однако сравнение этих символов с пробелами даст результат «не совпадает», так как в этих позициях содержатся символы, которые не имеют графического представления. Заметим, что встроенная функция TRIM отсекает с двух концов строки как пробелы, так и нулевые символы.

Символы, не имеющие графического представления. Каждый символ строки может иметь одно из 256 значений, а печатающие устройства нередко могут напечатать не более 100 различных символов. Оставшиеся символы относятся к символам, не имеющим графического представления.

Аналогия. *На электронных часах цифры формируются из нескольких светящихся сегментов (обычно их шесть или восемь). Из восьми горящих и потушенных сегментов можно составить 256 комбинаций, но только 10 из них осмысленны. При попытке напечатать на устройстве, которое может печатать только цифры, какую-нибудь комбинацию горящих сегментов, которая не является цифрой, устройство просто ничего не отпечатает, хотя эта комбинация и не является пробелом, которому соответствуют все потушенные сегменты. В памяти ЭВМ символы представляются в виде, очень напоминающем восемь зажженных и потушенных сегментов. При этом пробелу не соответствуют все потушенные сегменты (так же, как на часах, все потушенные сегменты - это не нуль). Символ, который распечатывается как восемь нулей - это так называемый пустой символ: он получается, если ЭВМ перед выполнением программы «чистит» память.*

Выполняемая в Windows ПЛ-программа всегда имеет обнуленную память в начале своей работы.

Если в программу вкралась непонятная ошибка и необходимо посмотреть, что же находится на месте символов, которые печатаются как пробелы, то можно применить следующий прием. Распечатка строки (или подстроки) оператором PUT EDIT(UNSPEC(СТРОКА))(B); будет печатать строку в виде последовательности нулей и единиц, где каждому символу строки соответствует восемь символов на печати. При этом пробелу соответствует комбинация «00100000».

Более точно такой прием позволяет напечатать символы в их двоичном коде: каждые восемь напечатанных таким образом цифр представляют собой код очередного символа. В справочниках приводятся таблицы кодов символов.

При печати строки переменной длины на ПЛ/1 следует помнить, что первый байт содержит текущую длину строки в двоичном представлении, т. е. код первого символа начинается с 9-го бита. Точный смысл слова UNSPEC выходит за рамки нашего рассмотрения.

Начальные присваивания строковым массивам. Строки без значений могут возникнуть из-за ошибок в начальных присваиваниях строковым массивам, связанных с ошибками в повторителях. Конструкция

```
DCL S(m) CHAR(n) INIT((3)'A');
```

может пониматься двояко: присваивание значения 'AAA' первому элементу массива или присваивание трех значений 'A' трем первым элементам. В ПЛ/1 принято первое толкование. Если же нужно получить эффект, который дало бы второе толкование, то надо написать так:

```
DCL S(m) CHAR(n) INIT((3)'A'(1));
```

В такой записи первый повторитель рассматривается как повторитель элементов, а второй - как повторитель символов в строковом значении.

В PL/1-KT повторитель символов строки пишется справа, а не слева от самой строки, например PUT SKIP LIST('='(80)); отчеркнет строку символом равенства.

Ясно, что игнорирование этого нюанса приводит к появлению строк без значения с возникновением всех описанных выше эффектов.

4.4. Сравнение строк

Общие правила. По правилам ПЛ/1 при сравнении строк разной длины более короткая строка дополняется пробелами (для символьных строк) или нулями (для битовых) до числа символов в более длинной строке, а затем выполняется поэлементное сравнение.

На первый взгляд эти правила достаточно естественны, и строка «АБВГ» получается меньше, чем строка «АБВГД», что соответствует нашим представлениям о лексикографическом порядке слов (т. е. порядке, как в словаре). Но если на битовых строках дополнение нулями никаких проблем не привносит, так как нуль меньше единицы, а других значений там быть не может, то при сравнении символьных строк могут возникнуть неожиданные эффекты, связанные с тем, что пробел - это не самый малый. символ. Символы упорядочены по некоторому алфавиту, например в кодировке MS DOS пробел занимает 32-е место, т. е. есть 31 символ меньше пробела. Все они относятся к символам, не имеющим графического представления, но могут попасть в строку, например, путем применения функции UNSPEC или наложения. Таким образом, при наличии в строках только алфавитно-цифровой информации со знаками препинания никаких проблем не возникает, но наличие символов, не имеющих графического представления, может дать неприятные эффекты при сравнении строк.

Лексикографическое сравнение по русскому алфавиту. Проблема работы со строками, связанная с тем, что алфавитная упорядоченность русских символов и коде EBCDIC не соответствует их упорядоченности в русском алфавите, ушла прошлое вместе с кодировкой ЕС ЭВМ.

Во всех кодировках, включая UNICODE, стало возможно прямое сравнение строк в лексикографическом порядке.

4.5. Преобразование строка-число

Назначение. В ряде случаев возникает необходимость получить строку, содержащую цифры числа, или, наоборот, из строки, содержащей цифры, получить их числовое представление. Формально ПЛ/1 позволяет использовать числовые выражения там, где нужна строка, и строковые выражения там, где нужно число. При этом происходят автоматические преобразования типов, которые использовать

не рекомендуется, ибо (как уже отмечалось ранее) они дают не всегда очевидный результат. Поэтому лучше использовать явные средства преобразования, которые также имеются в языке.

Внутренний ввод-вывод. Операторы GET и PUT могут использоваться с режимом STRING:

```
GET STRING(<строка>) EDIT (<список>) (<формат>);
PUT STRING(<строка>) EDIT (<список>) (<формат>);
```

или

```
GET STRING(<строка>) LIST (<список>);
PUT STRING(<строка>) LIST (<список>);
```

Эти операторы работают так же, как и операторы без режима STRING, но информация оператором GET берется не из файла SYSIN, а из указанной после STRING строки, а оператором PUT помещается не в файл SYSPRINT, а в указанную после STRING строку. Как и при обычном вводе-выводе, здесь происходит преобразование символьной информации в числовую.

Понятно, что в этих операторах в форматах нельзя использовать элементы SKIP и PAGE.

Например, после операторов

```
A=2.75; ...
PUT STRING(STR) EDIT(A)(F(7,3));
```

строка STR получит значение «__2.750», в то время как при попытке присвоить числовое значение обычным присваиванием STR=2.75; сработают автоматические преобразования типов и строка STR получит значение «_2.75000E+00».

Если строка S имеет значение « 123 'AAA' 2.750», то оператор

```
GET STRING(S) LIST(A,B,C);
```

введет значения 123, 'AAA' и 2.750, которые по правилам присваивания будут присвоены переменным A, B и C соответственно.

Глава 5. ОСОБЕННОСТИ ЧИСЛОВОЙ ОБРАБОТКИ

5.1. Формы представления и системы счисления

Представление действительных чисел в ЭВМ. В программировании между целыми и действительными числами имеется принципиальное различие: целые числа - точные, действительные - приближенные, причем с относительной погрешностью, ограниченной параметрами ЭВМ.

Действительные числа представляются в памяти ЭВМ в так называемой *форме с плавающей точкой* (отсюда и описатель FLOAT - плавающий), т. е. число x представляется в виде: a*n**b, где n - основание той системы счисления, в которой работает ЭВМ; a - называется мантиссой, содержит k n-ичных знаков и удовлетворяет соотношению 1/n < a < 1 (т. е. старший знак не нулевой) или же a=0; b - называется порядком и содержит q n-ичных знаков. Для процессоров x86 эти параметры таковы: n=16 (ЭВМ работает в шестнадцатеричной системе), k=20 (хранит двадцать цифр мантиссы), q=11 (т. е. порядок занимает одиннадцать полных цифр и еще 3 бита до знака). Для прояснения сути дела примеры будут рассмотрены применительно к гипотетической ЭВМ с параметрами: n=10, k=4, q=1, т. е. работающей в привычной нам десятичной системе счисления с короткими 4-разрядными числами (чтобы не проводить расчеты в шестнадцатеричной системе с длинными числами).

+2734.2=+0.27342.10**+4

+0.00273=+0.2730.10**-2

В памяти ЭВМ запоминается не само число, а только его мантисса со знаком и порядок со знаком; десятичная точка подразумевается перед первой цифрой мантиссы, а порядок показывает, куда и на сколько цифр «уплывает» точка на самом деле (отсюда и термин «плавающая точка»).

Диапазон и погрешность представления. Понятно, что в таком представлении чисел можно запомнить числа в диапазоне

$n^{-b_{max}-1} - n^{b_{max}-1}$

приблизительно (9.46*10**-308 , 1.18*10**308) на x86 и (10**-10, 10**9) на гипотетической ЭВМ), при этом, поскольку все цифры мантиссы, начиная с (k+1)-й, округляются, то относительная погрешность представления чисел равна n**(1-k)/2.

Все цифры n-ичные, отбрасываемые цифры таковы, что первая из них меньше n/2 (т. е. меньше 5 при n=10 и меньше 8 при n=16), иначе они округлялись бы до большего значения и последняя остающаяся цифра увеличивалась бы на единицу. Итак, относительная погрешность представляемых чисел в худшем случае равна половине предпоследнего разряда, т. е. равна 1/2*10**-3 для нашей гипотетической ЭВМ и равна например, 1/2*16**-5=0.5*10**-6 для ЕС ЭВМ.

Таким образом, особенности машинной арифметики - это работа в ограниченном диапазоне и с ограниченной точностью. Посмотрим, какие следствия из этого вытекают.

Работа в ограниченном диапазоне. Простейший пример: результаты операторов E=A*B*C; и E=A*C*B; могут не совпадать: при A=10**-40, B=10**-40, C=10**20 на ЕС ЭВМ первый оператор даст нулевой результат, а второй - правильный; при A=10**40, B=10**40, C=10**-20 первый оператор вообще не даст результат, так как промежуточное значение выходит за границы диапазона.

Работа с ограниченной точностью. Пусть на гипотетической ЭВМ вычисляется ($a_1+a_2+...+a_{1000}$) и пусть ($a_1+a_2+...+a_{900}$)=9555, а $a_{901}= a_{902}=…a_{1000}$=0.2. При сложении a_{901} с накопленной суммой получим 0.95552*10**4, но при запоминании этого результата ЭВМ потеряет последнюю цифру на округлении. В результате последние 100 компонент не внесут свой вклад в общую сумму, и накопленная погрешность будет весьма существенной (0.2*100=20.0). При этом, если отдельно просуммировать последние 100 компонент, а затем добавить полученную сумму к сумме первых 900 членов, получим правильный результат. Таким образом, в ЭВМ не выполняется ассоциативный закон. Не выполняется и дистрибутивный закон: на гипотетической ЭВМ получим (0.44440+0.00002)*5=2.222, а раскрыв скобки, получим 0.44440*5+0.00002*5=2.223.

5.2. Особенности машинной арифметики

Накопление погрешностей. Для того чтобы все же выполнять на ЭВМ расчеты и быть уверенным в точности результатов, надо знать, как распространяется и накапливается погрешность при выполнении операций. Ограничение имеется только на относительную погрешность, абсолютная погрешность может быть какой угодно: Dx=x*dx. При умножении и делении относительные погрешности умножаются, как и в обычной арифметике. При сложении и вычитании должны складываться абсолютные погрешности, но на ЭВМ это не так: сначала число с меньшей погрешностью приобретает погрешность того же порядка, что и число с большей погрешностью, а затем складываются уже получившиеся погрешности. Например (все дальнейшие примеры для гипотетической ЭВМ): число 10 и 1/3 представляется в ЭВМ в виде 0.1033*10**3, погрешность равна 0.0033...; число 1/3 представляется в виде 0.3333*10**0, погрешность 0.000033...; при сложении должно получиться 100 и 2/3, представляемое в ЭВМ в виде 0.1067*10**2 с погрешностью 0.0033..., но получим 0.1066*10**2, т. е. погрешность стала 0.0066... (рис. 5.2).

Иными словами, в худшем случае погрешность результата равна удвоенной погрешности максимального из чисел.

```
                                                    приобретаемая погрешность
   1/3=   0.3333*10**0 + 0.000033... 0.3333*10**0 => 0.003333*10**2
+
10 1/3=   0.1033*10**2 + 0.003333... 0.1033*10**2 => 0.1033*10**2
--------  ---------------------                     -----------------
10 2/3=   0.1067 10**2 - 0.003333...                   0.1066*10**2
                Исходные погрешности                Погрешность 0.0066
```

Рис. 5.2

Распространение погрешностей проиллюстрируем на простейшем примере. Пусть имеется цикл с параметром X, пробегающим значения от A до B с шагом H, причем значения A, B и H таковы, что N=(B-A)/H есть целое число. Сразу отметим, что «естественный» способ организации цикла

```
DO X=A BY H TO B; ... END;
```

может не дать желаемого результата: после N-кратного добавления H к значению X, которое изначально равно A, может получиться число чуть меньше или чуть больше B. Во втором случае цикл при последнем значении X уже работать не будет, т. е. сработает на один раз меньше, чем надо. Чтобы избежать этого, можно организовать цикл так:

```
DO X=A BY H TO B+H/2; ... END;
```

Но и этот вариант может не дать желаемого результата: при N-кратном сложении X с H может накопиться погрешность, превышающая H/2, и цикл не доработает один раз (или даже больше) и сработает лишний раз. Обеспечить нужное число повторений цикла можно таким способом:

```
N=(B-A)/H; X=A;
DO I=0 TO N;
. . . X=X+H;
END;
```

но при этом погрешность у X все равно накапливается, и последнее значение X может оказаться весьма далеким от B. Наилучшую точность ценой дополнительной операции умножения на каждом; шаге дает вариант

```
N=(B-A)/H;
DO I=0 TO N;
  X=A+I*H;…
END;
```

Продемонстрируем накопление погрешностей при разных методах организации цикла: пусть на гипотетической ЭВМ выполняются эти программы при A=100, B=103, H=1/3. Таблица 5.1 показывает, какие значения должно принимать X и какие получаются при разных вариантах программы.

Таблица 5.1. Накопление погрешностей

№ шага	X точное	X=X+H	X=A+I*H
0	100	100.0	100.0
1	100 1/3	100.3	100.3
2	100 2/3	100.6	100.7
3	101	100.9	101.0
4	101 1/3	101.2	101.3
5	101 2/3	101.5	101.7
6	102	101.8	102.0
7	102 1/3	102.1	102.3
8	102 2/3	102.4	102.7
9	103	102.7	103

Как видим, при последовательном добавлении H погрешность все нарастает и в итоге цикл сработает лишний раз. A в последнем варианте организации цикла округление происходит то в ту, то в другую сторону, и в итоге погрешность периодически компенсируется.

Пример расчета погрешностей. В общем случае расчет погрешностей, накапливающихся при работе по более или менее сложному алгоритму, представляет собой математическую задачу, требующую специального исследования. Мы рассмотрим этот расчет для случая цикла с действительным параметром, пример которого приведен выше. При проходе X от A до B с шагом H приходится выполнять [(B-A)/H]+1 сложений X с H (квадратные скобки означают целую часть, единица добавляется потому, что выход из цикла происходит, когда X переходит через B). Обозначим через p(z) порядок машинного представления числа z. Абсолютная погрешность X в худшем случае равна

D_x<p(x)*d<max { p(A),p(B)}*n**(1-k)/2,

где p(A),p(B) - это порядок A и B соответственно (n и k- параметры ЭВМ, соответственно, система счисления и длина мантиссы), погрешность H равна p(H)*d. При [(B-A)/H]+1 сложениях в худшем случае получаем суммарную погрешность

([(B-A)/H]+1)*max{p(A),p(B),p(H)}*n**(1-k)/2

(на каждом сложении к X может добавиться максимальная из погрешностей X и H). Чтобы цикл работал правильное число раз, накопление погрешностей не должно превышать H/2 (цикл организован до B+H/2). Но это ограничение оказывается слишком жестким, так как предполагает, что на каждом сложении мы имеем наихудший случай, чего реально не бывает никогда: по этой формуле даже цикл от 100 до 101 с шагом 1/3, должен на гипотетической ЭВМ идти неверно. Из эвристических соображений (даже в таком простом случае приходится привлекать эвристику!) можно ослабить ограничения вдвое

([(B-A)/H]+1)*max{p(A),p(B),p(H)}*1/2*n**(1-k)<|H| (1)

На практике достаточно считать, что при выполнении ограничения (1) можно ограничивать цикл последовательным наращением X. Если же это ограничение не удовлетворяется, то возможно, что цикл сработает нужное количество раз, но последнее значение X будет скорее всего весьма далеким от B. Рассмотренный выше числовой пример показывает, что при A=100, B=101, H=1/3 соотношение (1) истинно (0.2<1/3) и цикл идет правильно, при B=102.4 и тех же A и H соотношение ложно (0.4>1/3), но цикл все же идет нужное число раз, хотя последнее значение ближе к B-H, чем к B, а при B=103 соотношение ложно (0.5>1/3,) и цикл делает лишний шаг.

При организации цикла с вычислением X на каждом шаге по формуле X=A+I*H погрешность X должна быть меньше H/2, чтобы не путать соседние значения X, а значит удовлетворять соотношению

2{max{p(A)*d,p(I*H)*d}<max:{p(A),p(B+H-A)}*1/2*n**(1-k)}<|H|/2. (2)

Здесь никаких ослаблений из эвристических соображений делать нельзя, так как значение X получается всего двумя операциями, и возможность иметь на каждой операции худший случай вполне реальна (шанс иметь худший случай на каждом из десятка добавлений H к X невероятен). Соотношение показывает, что от 100 до 103 с шагом 1/3 пройти можно. Так же, как и для предыдущего способа организации цикла, невыполнение соотношения еще не гарантирует ошибки, но показывает, что шансы на нее велики. Например, несмотря на невыполнение соотношения (2) при A=101, B=102, H=0.1 (получаем 0.1>0.05), цикл сработает правильно, но уже при H=0.09 получим, что A+5*H и A+6*H, равные 101.45 и 101.54 соответственно, для ЭВМ неразличимы. Итак, при невыполнении соотношения (2) у программиста нет никаких средств заставить переменную X пробежать нужное ему множество значений.

Выводы. Прежде, чем программировать численный алгоритм, программист должен:

а) *убедиться, что для данного алгоритма и используемой ЭВМ все промежуточные данные оказываются в пределах допустимого ЭВМ диапазона, а вносимые машинной арифметикой погрешности не выходят за допустимые пределы*;

б) если вариант а) не имеет места, *то можно проверить то же при дополнительных ограничениях на исходные данные (установить, что на ваших данных все будет в порядке, хотя в общем случае этого гарантировать нельзя).*

Вариант а имеет место для формулы $(a_1+a_2,+...+a_n)/n$ применительно к переполнению: при работе по этой формуле оно произойти не может, но от исчезновения мы не гарантированы. Чаще реализуется вариант б: он позволяет работать по формуле $(a_1+a_2+.-.+a_n)/n$, если $(a_1+a_2+.-.+a_n)$ лежит в пределах допустимого диапазона, и организовывать цикл от A до B с шагом H в большинстве случаев обычным способом.

Если и вариант б не имеет места, то приходится выполнить следующее:

в) *разработать алгоритм, компенсирующий особенности машинной арифметики,* или

г) *использовать вычисления с увеличенным диапазоном и/или точностью* (реализованные на ЭВМ аппаратно или программно, а в языке - языковыми средствами или подпрограммами), в частности, *выполнить программу на другой ЭВМ с более точной арифметикой.*

Фактически до сих пор мы разрабатывали компенсирующие алгоритмы: изменение порядка умножений в формуле A*B*C избавляло нас от переполнения или исчезновения, причем без дополнительных затрат; заменой $(a_1+a_2+...+a_n)/n$ на $(a_1/n+a_2/n+...+a_n/n)$ уходили от переполнения ценой дополнительных действий ценой дополнительных действий организовывали цикл наиболее точным способом.

Отметим, что выполнение вариантов в или г все равно требует выполнения проверок а или б, чтобы убедиться в достижении искомого результата.

Компенсирующие алгоритмы. В качестве примера наиболее часто встречающегося компенсирующего алгоритма отметим, что *действительные числа никогда нельзя сравнивать на равенство*

```
IF A=B THEN ...
```

ибо из-за погрешности их вычисления ЭВМ почти со 100%-ной вероятностью ответит «не равны»; вместо этого следует писать

```
IF ABS(A-B)<EPS THEN …
```

где EPS имеет какое-то малое значение (скажем, 0.00000001), т.е. действительные числа считаются равными, если они различаются не более, чем на некоторое число E. Кроме того, вычисления следует стараться организовать так, чтобы по возможности удерживать промежуточные результаты в середине допустимого диапазона (см. пример с умножением A*B*C).

Для работы в расширенном диапазоне или с повышенной точностью в ПЛ/1 также предусмотрены некоторые средства: можно описывать переменные с атрибутами BINARY FIXED(31) и FLOAT DECIMAL(16). Числа с атрибутом BIN FIXED (или что то же самое, с атрибутом BIN FIXED(15)) имеют диапазон до 2**15=32768, числа с атрибутом BIN FIXED(31) имеют диапазон до 2**31 =10**10*2. Числа с атрибутом FLOAT (или что то же самое, с атрибутом FLOAT DECIMAL(6)) имеют точность около 10**-6, а числа с атрибутом FLOAT DECIMAL(16) имеют точность около 10**-16. Переменная с атрибутами BIN FIXED(31) занимает 4 байта памяти (в 2 раза больше, чем BIN FIXED), но обрабатывается практически с той же скоростью. Переменная с атрибутом FLOAT (16) занимает 8 байт памяти (в 2 раза больше) и обрабатывается тоже с такой же скоростью.

Типовые ошибки определения точности. Первая типовая ошибка связана с особенностями именно языка ПЛ/1 (все остальное, что относится к точности, от языка программирования не зависит):

Запись DCL K BIN FIXED;…Z=2.1*K;

предполагает, что целое значение К будет умножено на действительное значение 2.1. Однако с точки зрения языка ПЛ/1 константа «2.1» - это не действительное число (строго говоря, в ПЛ/1 вообще нет понятия «действительное число» в том смысле, как оно здесь понимается), поэтому при умножении эта константа сначала преобразуется к нужному виду по тем правилам, которые уже упоминались, при этом она получит значение, равное 2.0625 (существенная потеря точности), потом выполняется умножение, затем результат с внесенной таким образом погрешностью преобразуется к типу, соответствующему описанию переменной 2. Заметим также, что если вместо 2.1 взять константу 2.1000, то после преобразования получим значение 2.09998 (уже лучше), если же написать 2.1E0 или 0.21E1 (любой вариант, в котором присутствует «Е» и порядок), то в расчете будет участвовать значение 2.099999 (это наилучший вариант, ибо 2.1 в шестнадцатеричной системе точно не представляется). Чтобы не вникать в эти правила преобразования, но и не наталкиваться на вносимые ими погрешности, лучше приучиться не использовать в выражениях дробные константы, а заводить специальные переменные, куда эти константы помещать с помощью начальных присваиваний, и в выражениях использовать эти переменные:

DCL K BIN FIXED, M FLOAT INIT(2.1E0); …Z=M*K;

На целых константах таких проблем не возникает.

Вторая типовая ошибка иллюстрируется примером

Z=3.14*COS(2*X);

в котором константа 3.14 призвана изображать математическую константу «пи». Но эта константа задает «пи» с огромной погрешностью 0.00159..., которую мы

запускаем в наш расчет. Трудно предсказать, во что вырастет эта погрешность, пройдя через расчет.

Здесь также надо завести переменную, в которую записать значение константы с максимально возможной точностью, а затем вместо длинного значения использовать более короткий идентификатор.

В PL/1-KT имеется специальные функции ?PI и ?PI2 возвращающие значение «пи» как FLOAT DECIMAL(6) и FLOAT DECIMAL(16) соответственно. Они достают значение из прошитой константы процессора x86.

Последняя ошибка связана с тем, что при вычислениях с заданной точностью надо обеспечивать соответствующую точность у всех промежуточных результатов. Например, во фрагменте

```
DCL (A, B) FLOAT, C FLOAT (16); ... C=A*B;
```

мы не получим двойную точность у C: результат A*B будет иметь обычную точность, затем при присваивании он будет преобразован к двойной точности путем добавления нулей в конец мантиссы.

Чтобы получить правильный результат с двойной точностью, надо написать:

```
DCL (A,B) FLOAT, (AT,BT,C) FLOAT(16);
AT=A; BT=B; C=AT*BT;
```

При копировании A и B в AT и BT значения будут преобразованы к двойной точности путем добавления нулей к мантиссам, а умножение уже будет выполняться с двойной точностью. В данном примере достаточно было скопировать одну переменную, вторая при этом была бы преобразована к двойной точности автоматически, но в операторе `C=A*B*AT;` при тех же описаниях первое умножение выполнится с обычной точностью, так как оба операнда - с обычной точностью, а перед вторым умножением результат первого умножения будет преобразован к двойной точности путем приписывания нулей.

5.3 Работа в сверхдиапазоне

Метод постоянных масштабов. Иногда возникает необходимость выполнить расчеты с числами, по диапазону или требуемой точности превосходящими те границы, которые предоставляет нам язык или ЭВМ. В этом случае можно выйти из положения следующим способами.

Когда значения выходят за допустимый диапазон, но понятно, как изменится результат, если исходные данные уменьшить или увеличить в несколько раз, можно применить *метод постоянных масштабов*. Например, пусть значения некоторой переменной лежат в диапазоне 10**-3-10**-1, и в расчете, выполняемом на ЭВМ, приходится использовать степени этой переменной вплоть до 40-й. Ясно, что (10**-3)**40 выходит за диапазон ЭВМ. Но если по физическому смыслу расчета видно, что при увеличении этого значения в 100 раз (т. е. при задании его не в метрах, а в

сантиметрах) результат увеличится в 10000 раз, то расчет выполняется с увеличенным в 100 раз значением, степени которого теперь оказываются в диапазоне 10**-40...10**40, а затем результат уменьшается в 10000 раз, что можно сделать непосредственно на выводе, не меняя значения в памяти ЭВМ: PUT EDIT(A*1E-4)(F(8,5));

Не обязательно масштаб должен быть десятичным. Если при тех же условиях значения лежат в диапазоне 10**-3…10**0 (т. е. изменяются от тысячных долей до единиц), то при масштабе 10 они приводятся к диапазону 10**-2…10**1 и получаем исчезновение на нижней границе диапазона: (10**-2)**40 < 10**-78, а при масштабе 100 они приводятся к диапазону 10**-1…10**2 и получаем переполнение на верхней границе диапазона: (10**2)**40 > 10**75. Но можно взять масштаб 2**5 (вводим новую единицу измерения, равную 32 сантиметрам), и данные окажутся в диапазоне (2**5)*10**3...2**5, а 40-я степень данных - в диапазоне около 10**-60…10**60, и расчет можно выполнять.

Метод плавающих масштабов. Если метод постоянных масштабов неприменим, то можно использовать *метод плавающих масштабов*, при котором каждое значение изображается двумя переменными: ZX и MX, в совокупности представляющими значение ZX*10**MX. Сложение значений выполняется подпрограммой

```
SLOZH:PROC(MA,ZA,MB,ZB,MC,ZC);        /* C=A+B */
DCL
(MA,MB,MC,MD) BIN FIXED,
(ZA,ZB,ZC,ZD)    FLOAT DEC(16);

IF MA>MB
   THEN DO; ZC=ZA; MC=MA; ZD=ZB; MD=MB; END;
    ELSE DO; ZC=ZB; MC=MB; ZD=ZA; MD=MA; END;

ZD=ZD* 10** (MD-MC);
ON OFL BEGIN; MC=MC+1; ZC=ZC/10; ZD=ZD/10; GOTO C; END;
C:ZC=ZC+ZD;
END SLOZH;
```

а умножение – подпрограммой

```
UMN:PROC(MA,ZA,MB,ZB,MC,ZC);        /* C=A*B */
DCL
(MA,MB,MC) BIN FIXED,
(ZA,ZB,ZC)   FLOAT DEC(16);

ON OFL BEGIN;
    IF ZA>ZB THEN DO; ZA=ZA/10; MA=MA+1; END;
             ELSE DO; ZB=ZB/10; MB=MB+1; END;
    GO TO C;
END;
ON UFL BEGIN;
    IF ZA<ZB THEN DO; ZA=ZA*10; MA-MA-1; END;
             ELSE DO; ZB=ZB*10; MB=MB-1; END;
    GO TO C;
END;
C: ZC=ZA*ZB; MC=MA+MB;
END UMN;
```

Вычитание и деление выполняется аналогичными подпрограммами. Алгоритм сложения основан на том, что число с меньшим масштабом приводится к большему масштабу, причем если различие в масштабах очень велико, то число с меньшим масштабом заменяется нулем. Далее выполняется обычное сложение, и если в результате получается слишком большое число, то у слагаемых увеличивается масштаб, сами слагаемые уменьшаются и сложение повторяется. Умножение

выполняется по простейшему алгоритму: собственно значения умножаются, а масштабы складываются. При этом, если возникает переполнение (исчезновение), то число с большим (меньшим) значением изменяет свой масштаб так, чтобы оказаться ближе к середине допустимого диапазона. Название «метод плавающих масштабов», происходит оттого, что масштабы чисел меняются («плавают») в процессе счета. Эти подпрограммы призваны только проиллюстрировать идею, поэтому они реализованы простейшим и весьма далеким от оптимальности методом (с учетом особенностей работы ЭВМ выгоднее было взять шестнадцатеричный, а не десятичный масштаб, удерживать значения ближе к середине диапазона и т. д.).

5.4. Работа со сверхточностью

Представление чисел. Для работы со сверхточностью каждое число следует представлять несколькими переменными, например массивом, описанным DCL A(-2:1) FLOAT DEC(16); где A(K)<M=10000000 и все A(K) имеют одинаковый знак. Такой массив представляет число, равное A(1)*M+A(0)+A(-1)*M**-1+A(-2)*M**-2. Сложение и умножение чисел реализуется подпрограммами (которые также написаны по принципу достижения максимальной простоты и краткости текста без заботы об оптимальности по скорости).

Алгоритм сложения. Фактически число записывается в системе счисления с основанием 10000000, и элементы массива хранят «цифры» этого числа. При сложении «цифры» складываются «в столбик», если получающаяся «цифра» превосходит M (равное 10000000), то выполняется перенос в следующий разряд. При сложении чисел с разными знаками фактически происходит вычитание «цифр» и при необходимости приходится «занимать» из соседнего разряда. Некоторое удлинение программы сложения происходит из-за того, что у отрицательных чисел могут быть нулевые «цифры», которые имеют в ЭВМ положительный знак. Если старшая «цифра» получилась большей M, то имитируется переполнение, которое другим путем произойти не может:

```
SLOZH:PROC(A,B,C)        /* C=A+B */
DCL
((A,B,C)(-2:1),
  D(-4:2),
  P,
  M STATIC INIT(10000000)
)FLOAT DEC(16),
(I,K) BIN FIXED;

C=A+B;                        /* СЛОЖЕНИЕ */;
DO I=1 BY -1 TO -2 WHILE(C(I)^=0); END;

IF I>= -2 THEN               /* ПРИВЕД. К ОДИНАКОВЫМ ЗНАКАМ */
  DO; IF C(I)<0 THEN S= -1; ELSE S=1;
     DO K=-2 TO I-1;
       IF C(K)*S<0 THEN
          DO; C(K+1)=C(K+1)-S; C(K)=C(K)+M*S; END;
     END;
     IF ADS(C(I))>=M THEN SIGNAL OVERFLOW;
   END;
END SLOZH;
```

Алгоритм умножения. При перемножении «цифра» при k-й степени M у результата равна сумме произведений «цифр» при i-й степени M у одного сомножителя и j-й - у другого, где i+j=k. Со знаками у «цифр» все получается автоматически. «Цифры» при M**-3 и M**-4 округляются, остальные разделяются на итоговую «цифру» и единицу переноса в следующий разряд. Так как все «цифры» меньше 10**7, то переполнения при вычислениях быть не может (максимальный коэффициент может быть при M**-1, но он меньше 4*10**14). Проблем с точностью не возникает: как показывает рис. 5.1, при n=16, k=14 целые числа будут представляться в памяти точно в диапазоне до n**k.

```
UMN:PROC(A,B,C);                /* C=A*B */
DCL
((A,B,C)(-2:1),D(-4:2),P,M STATIC INIT(10000000)
)FLOAT DEC (16),
(I,J) BIN FIXED;

D=0;                            /* ПЕРЕМНОЖЕНИЕ */
DO I=-2 TO 1;
  DO J=-2 TO 1;
    D(J+J)=D(J+J)+A(I)*B(J);
  END;
END;
                                /* ПЕРЕНОСЫ*/
P=TRUNC( (D(-3)+TRUNC(D(-4)/M))/M-0.5);
DO I=-2 TO 1;
 C(I)=D(I)+P; P=TRUNC(C(I)/M); C(I)=C(I)-M*P;
END;
IF P+D(2)^=0 THEN SIGNAL OVERFLOW;
END UMN;
```

Ввод-вывод. Печать этих чисел с использованием рассмотренных средств языка ПЛ/1 выполняется подпрограммой

```
PECHAT:PROC(A);             /* ПЕЧАТЬ */
DCL
 A(-2:1)FLOAT DEC(16), S CHAR(1), I BIN FIXED;
S='+';
DO 1= -2 TO 1; IF A(I)<0 THEN S='-'; END;
PUT EDIT (S,A(1),A(0),'.',A(-1),A(-2)) (A,2 P'9999999',A, 2 P'9999999');
END PECHAT;
```

Можно усложнить подпрограмму, чтобы она подавляла печать незначащих нулей. (Попутно отметим, что в описателе P'9…9' нельзя задавать более 15 цифр.)

Для ввода несложно написать аналогичную подпрограмму.

Итоги. В результате получаем возможность работать с 28-разрядными числами в диапазоне 10**-14...10**14. При необходимости можно объединить приемы работы в сверхдиапазоне и со сверхточностью. В заключение отметим, что сверхдиапазон и сверхточность оказываются необходимыми гораздо чаще, чем того хотелось бы программистам.

Глава 6. БЛОКИ И ПОДПРОГРАММЫ

6.1. Блочная структура программы

Назначение. Блоки являются очень удобным средством выделения памяти по ходу выполнения программы и ее освобождения по мере исчезновения надобности в ней. Использование блоков позволяет получить программу с меньшими требованиями к оперативной памяти и обеспечить удобство написания частей программы без заботы о том, не совпали ли идентификаторы переменных в одной части с идентификаторами в другой. К сожалению, программисты недостаточно активно пользуются средствами, предоставляемыми блочной структурой.

Динамическая настройка по размерам массивов. Очень многие алгоритмы используют массивы, размеры которых являются параметром алгоритма. Естественно строить программы, реализующие эти алгоритмы так, чтобы они могли обработать данные любых размеров, которые в принципе помещаются в памяти. Наиболее распространенный метод решения этой проблемы, применяемый в языках без блочной структуры, - это статическая настройка по параметрам, заключающаяся в изменении констант в теле программы с последующей ее перетрансляцией:

```
P:PROC OPTIONS(MAIM);
%REPLACE N BY 10000;
DCL A(N) FLOAT;
…
DO I=1 TO N;
  <обработка элемента A(I)>
END;
…
END;
```

Понятно, что перетрансляция приемлема, если изменения N происходят редко. Если же для каждого нового выполнения надо менять N, то N задается в исходных данных. Получаем *динамическую настройку* (т. е. настройку в процессе выполнения без перетрансляции программы) по размеру N. для чего нередко изменяют прием, называемый *работой с неполным массивом*, при которой массив должен быть описан с запасом, а используется только его часть:

```
P:PROC OPTIONS(MAIN);
DCL N BIN FIXED, A(10000) FLOAT;
GET LIST(N);
GET LIST((A(I) DO I=1 TO N);
…
DO I=1 TO N;
   <обработка элемента A(I)>
END;
…
END;
```

Этот метод также имеет свои ограничения: не всегда известен размер запаса, кроме того, при превышении размерами массива допустимого запаса все равно приходится перетранслировать программу с измененным размером массива. Полная динамическая настройка программы по размерам массивов дает программе возможность обрабатывать массивы любых размеров, лишь бы они поместились в память:

```
P:PROC OPTIONS(MAIN);
DCL N BIN FIXED;
DCL P PTR;
DCL A(1000000) FLOAT BASED(P);
GET LIST(N);
ALLOCATE (4*N) SET(P);
DO I=1 TO N; GET LIST(A(I)); END;
DO I=1 TO N;
<обработка элемента A(I)>
END;
…
END;END;
```

Иногда ценой некоторого усложнения программы можно даже избавить пользователя от необходимости задавать N. заставив саму программу подсчитывать его:

```
P:PROC OPTIONS(MAIN);
DCL N BIN FIXED, A FLOAT;
N=0;
ON ENDFILE(SYSIN) GO TO R;
C:GET LIST(A); N=N+1; GO TO C;
R:CLOSE FILE(SYSIN); OPEN FILE(SYSIN);

BEGIN;
DCL P PTR;
DCL A(1000000) FLOAT BASED(P);
ALLOCATE (4*N) SET(P);
DO I=1 TO N; GET LIST(A(I)); END;
DO I=1 TO N;
<обработка элемента A(I)>
END;
…
END;
```

В этой программе работа начинается с того, что исходные данные считываются все в одну переменную, причем каждое следующее значение стирает предыдущее. Основная задача этого цикла - подсчитать число исходных данных. Выход из цикла ввода происходит по исчерпанию файла данных. Закрытие и повторное открытие файла приводят к тому, что читаться он будет с начала. Теперь можно войти в блок, отвести для массива A, столько памяти, сколько ему надо, и работать с этим массивом.

Если требуется ввести таким способом несколько массивов, то они должны вводиться из разных файлов.

Экономия памяти. В ряде случаев программу можно разделить на несколько частей, где в одной части нужен массив A, а в другой массив A уже не нужен, но нужен массив B. При значительных размерах этих массивов может возникнуть проблема экономии памяти. Простейшее решение этой проблемы дает блочная структура программы.

```
P:PROC OPTIONS(MAIN);
DCL C(1:100) FLOAT;
DCP P PTR;
ALLOCATE (100000) SET(P);
BEGIN; DCL A(20000) BIN FIXED BASED(P);
  <работа с массивами A и C>
END;
```

```
BEGIN; DCL B (50000) FLOAT;
  <работа с массивами B и C>
END;
END;
```

Обычно в подобных случаях в программе имеются какие-то переменные, описанные во внешнем блоке, через которые части программы обмениваются информацией (в данном примере это массив C). При работе первого вложенного блока обрабатываются массивы A и C, результаты остаются в C. При работе второго вложенного блока обрабатываются массивы B и C, причем из C берутся данные, оставленные там первой частью.

Вспомогательные массивы в подпрограммах. Некоторые алгоритмы для своей работы требуют вспомогательную память в количестве, зависящем от размеров массивов-параметров. Например, подпрограмма решения системы линейных уравнений по некоторым методам имеет в качестве входных параметров матрицу системы A(1:n,1:n) и массив свободных членов B(1:n), решение получается на месте свободных членов, а в процессе работы ей нужен вспомогательный массив размера n. В языках без блочной структуры (в ФОРТРАНе, например) такой массив приходится передавать через параметр дополнительно к основным параметрам, что нелогично, ибо внешней программе этот массив ни к чему, либо же внутри подпрограммы приходится описывать массив постоянного размера с запасом, расходуя память и рискуя получить неопределенные эффекты, если фактические параметры превысят максимально допустимое значение.

Но процедурный блок также является блоком, и при входе в него можно запросить память, в том числе и для массивов с переменными границами, что позволяет писать универсальные процедуры. Например, заголовок и описания процедуры для решения системы линейных уравнений могут иметь следующий вид:

```
LINUR:PROC (PA,PB,N,ERR);
DCL
(PA,PB,P1) PTR,
N BIN FIXED;
A(10000000)     FLOAT BASED(PA)
B (10000)       FLOAT BASED(PB),
C (10000)       FLOAT BASED(P1),
ERR LABEL;
ALLOCATE(4*N) SET (P1); …. FREE P1->C;
```

Эта процедура может получить массивы A и B произвольных размеров (требуется только, чтобы матрица A была квадратной, с диапазоном индексов от единицы и у массива B был диапазон индексов такой же, как у строк матрицы A) и метку ERR (выход по которой происходит при отсутствии решения у системы). Работа

процедуры начинается с того, что она запрашивает себе память для массива N с тем же диапазоном индексов, что и у массива B. В массиве C запоминаются перестановки столбцов, которые производятся по одному из алгоритмов решения системы, чтобы в конце вернуть столбцы на свои места. По окончании работы процедуры эта память освобождается.

В итоге получаем процедуру, которая может обработать данные любых размеров, лишь бы они поместились в оперативной памяти. При этом детали внутреннего устройства процедуры, в том числе требования к памяти, скрыты в ней полностью.

6.2. Ошибки в передаче параметров подпрограмм

Несовпадение размеров массивов. В языке ПЛ/1 почти не контролируется правильность передачи параметров, и вся ответственность за это возлагается на программиста. Именно он должен заботиться о соответствии количества и типов формальных и фактических параметров. В PL/1-KT имеется утилита LINT для контроля соответствия параметров.

Рассмотрим внешнюю процедуру S с описаниями

```
S:PROC (A);
DCL A(100) FLOAT;
...
END:
```

предположим, что в головной программе она оказалась по ошибке описанной так:

```
DCL S ENTRY((90) FLOAT);
```

Кроме того, в головной программе имеются следующие фрагменты:

```
GOL:PROC OPTIONS(MAIN);
DCL A(90) FLOAT, C CHAR(100) STATIC, B(20) BIN FIXED;
DCL S EMTRY((90) FLOAT);
...CALL S(A);
...
END;
```

При обращении к процедуре ей сообщается только адрес массива, а все остальное процедура определяет по своим описаниям.

Таким образом, с ее точки зрения этот массив состоит из 100 элементов и начинается там же, где массив A головной программы, т. е. в него кроме 90 элементов из A входят еще и 10 элементов, размещенных вслед за A.

Чаще всего транслятор размещает данные в том же порядке, в котором они заданы в описаниях, тогда вслед за массивом A располагается массив C и процедура захватит элементы C. Но это еще зависит от класса памяти (AUTOMATIC, STATIC) и вслед за массивом A окажется размещенным массив B.

Таким образом, даже не всегда очевидно, что будет испорчено в результате этой ошибки.

Пока процедура работает в пределах первых 90 элементов, ошибка заметна не будет; если же обработка затронет последние 10 элементов массива A, то фактически будут захвачены элементы массива B или C, если же процедура еще и меняет значения в массиве A, то «неожиданно» будут испорчены значения массива B или C.

Наиболее интересен этот эффект на многомерных массивах. Если процедура получает неправильную информацию о размерах массива, например, таким образом:

```
GOL:PROC OPTIONS(MAIN);
DCL A(3,4) FLOAT, P ENTRY((3,4) FLOAT), ...
... CALL P(A); ...
END;
P:PROC(B);
DCL B(2,3) FLOAT,
...
END;
```

то процедура будет считать, что ей дали массив из шести элементов и эти элементы имеют индексы, не совпадающие с индексами, которые они имеют на самом деле (рис. 6.1).

Индексация с точки зрения головной программы,

A_{11} A_{12} A_{13} A_{14} A_{21} A_{22} A_{23} …

B_{11} B_{23} B_{13} B_{21} B_{22} B_{23} B_{31} …

Индексация с точки зрения процедуры

Рис. 6.1

Рекомендации для избегания подобных ошибок просты: никогда не описывать массивы-параметры с константами в размерах массива и использовать функции LBOUND, HBOUND, DIMENSION.

Ошибки несовпадения размерности. Если процедура требует, например, двумерного массива, а ей дали одномерный, то, как правило, работа пойдет с эффектами не присвоенных значений или «защита памяти».

Ошибки несовпадения типов. Следующая ошибка связана с неправильной информацией о типе параметров. Передача параметров в ПЛ/1 не предполагает передачи никакой информации о типе фактического параметра. Например, процедура считает, что ей дают действительные значения в массиве, а на самом деле получает целые. В этом случае эффект не определен, однако можно описать

наиболее часто встречающиеся эффекты, которые можно понять, исходя из самых общих принципов кодировки информации в памяти ЭВМ.

Данные типа BIN FIXED записываются в памяти просто в виде числа со знаком в шестнадцатеричной системе (что для понимания сути дела не важно). Условно можно принять, что они содержат пять десятичных цифр. Например, число 7 записывается в виде «+00007». Данные типа FLOAT записываются в нормализованном виде с порядком, что описано в гл. 5. Они занимают примерно 10 цифр, из которых первые две - порядок (шестнадцатеричное представление для понимания сути дела не важно). Порядок записывается со смещением, т. е. к нему прибавляется некоторая величина, чтобы он всегда получался положительным. Для объяснения сути дела будем считать, что смещение равно 50, т. е. записывается число «50+порядок», в итоге числа большие единицы имеют порядок больший 50. Таким образом число 7.0E0, в нормализованной форме равное 0.7E1, записывается в виде «+5170000000» («51» - порядок, «70000000» - мантисса). Несмотря на некоторое отступление от истинной формы представления данных в памяти ЭВМ, эти сведения позволяют понять эффекты ошибок в типах параметров. Если вместо параметра типа FLOAT процедура получает значение типа BIN FIXED, то она берет два байта памяти, отведенные этому значению, и следующие два байта. Первые две цифры значения интерпретируются как порядок. Сравнивая представление в памяти целого значения 7 и действительного значения 7, видим, что целое значение будет проинтерпретировано как 0.007...E-50, т. е. как исчезающе малое число, к тому же не нормализованное. Чаще всего работа с ним начинается с попытки нормализовать его, что немедленно дает ситуацию UNDERFLOW.

Обратная ошибка - вместо целого значения получили действительное. Здесь обрабатываются только первые два байта значения, но цифры порядка воспринимаются как значащие цифры, в итоге значение расшифровывается как какое-то очень большое число (в нашей аналогии +51700).

Если же формальный параметр имеет атрибуты DEC FIXED, а фактически получено значение типа FLOAT или BIN FIXED, то наиболее вероятно возникновение ошибки «данные», так как данные типа DEC FIXED имеют своеобразный способ кодирования в памяти ЭВМ, и некоторые значения типа FLOAT или BIN FIXED просто не могут быть расшифрованы как данные типа DEC FIXED.

Обратная ситуация - DEC FIXED вместо FLOAT или BIN FIXED - с большой вероятностью приведет к ситуации случайных чисел для FLOAT к тому же ненормализованных.

Аналогия. *Когда мы записываем показание времени 3.5, то может подразумеваться: 3 часа 5 минут, 3 с половиной часа, 3 минуты 5 секунд, 3 с половиной минуты и т. д. Если сообщить только число, не указав единицы*

измерения, т. е. не указав способ кодирования, то человек будет считать, что используется привычный для него способ (метод кодирования определяется по своим описаниям), если же тот, кто сообщил таким образом время, пользовался другим способом кодирования, то возникает неопределенный эффект. Для выяснения того, что именно получится, надо знать оба способа кодирования (т. е. особенности представления в памяти ЭВМ). Для кодирования показаний электронных часов можно занумеровать каким-то способом светящиеся сегменты и записывать номера зажженных сегментов. Если по этой записи мы попытаемся установить время на обычных стрелочных часах, где время кодируется двумя числами: углом поворота часовой и минутной стрелок, то номера сегментов поставят нас в тупик - это и можно диагностировать как «данные».

Таким образом, если изучить кодирование данных разных типов у памяти ЭВМ, то можно предсказать результаты любых ошибок.

Ошибки количества параметров. Последняя ошибка связана с количеством параметров. Если в вызывающей программе и процедуре имеются противоречия относительно числа параметров, например, подпрограмме дали лишний параметр или подпрограмме недодали один параметр, то получаем следующие эффекты. В первом случае процедура просто не будет использовать значение «лишнего» параметра и никаких внешне сразу заметных эффектов возникать не будет. Во втором случае практически неизбежны ситуации «защита памяти», реже – «данные», а если недостающий параметр - имя подпрограммы, то - ситуация «операция». Дело в том, что в ПЛ/1 всегда в , качестве параметра передается его адрес (место размещения в памяти фактического параметра). Понятно, что попытка обратиться в память по случайному адресу даст именно эти эффекты.

6.3. Подпрограммы-функции

Тип возвращаемого результата. Выходным параметром функции является ее идентификатор. Именно к нему относится описатель RETURNS(...) в заголовке процедуры, определяющий тип значения, который получает идентификатор функции. Строго говоря в ПЛ/1 нет различий между процедурами и функциями. В PL/1-KT использование функций как процедур определяется типом возвращаемого значения. Значения типа DEC FIXED, FLOAT и CHAR передаются через стек и попытка выполнить эти функции как процедуры вероятно приведет к ситуации «операция». Значения типа BIN FIXED и BIT передаются через регистры и такие функции можно использовать как процедуры.

Если формально в программе возможен выход из функции не оператором RETURN, а достижением последнего END тела функции – транслятор выдает предупреждение.

К функции (возвращающей BIN FIXED или BIT) можно обратиться как к процедуре оператором CALL, в результате вычисленное значение функции теряется, так как оно не используется в операторе CALL. Если функция не дает побочных эффектов, то такое обращение равносильно пустому оператору.

Несоответствие типа возвращаемого значения, описанного в DCL-ENTRY, и реального типа возвращаемого значения приводит к таким же ошибкам, как и несоответствие типов параметров процедур, ибо имя функции - это такой же параметр. Например, пусть имеются описания:

```
GOL:PROC OPTIONS(MAIN);
DCL S ENTRY(FLOAT,BIN FIXED) RETURNS(BIN FIXED);
…
Z=S(A,5); ...
END;

S:PROC 5(A,B) RETURNS(FLOAT);
DCL A FLOAT, B BIN FIXED,...;
RETURN(..);
END;
```

Головная программа сочтет, что получает значение типа BIN FIXED, и будет расшифровывать результат как значение типа BIN FIXED со всеми описанными выше эффектами.

Побочный эффект. Поскольку свой единственный результат функция присваивает своему идентификатору, то все остальные параметры должны быть входными. Однако ЭВМ это не контролирует и, если операторы функции осуществляют какие-то изменения в значениях параметров, возникает так называемый *побочный эффект*, т. е. эффект, производимый функцией помимо своей основной работы - вычисления значения. Хотя можно найти примеры, когда побочный эффект может принести пользу, чаще он вреден; хорошим стилем программирования считается отсутствие побочных эффектов.

В последнее время побочный эффект поставлен что называется «с ног на голову», например все процедуры (API) Windows возвращают значение – код ответа. Здесь побочный эффект является главным, а возвращаемое значение необязательным и может вообще не использоваться.

Рассмотрим функцию, вычисляющуюК!:

```
FAKT: PROC (К) RETURNS(BIN FIXED);
DCL (К,F) BIN FIXED;
F=1; DO К=К ВУ -1 TO 2; F=F*К; END;
```

```
RETURN(F); END;
```

Эта функция реализована очень экономно: в качестве параметра цикла использован ее входной параметр. Значение функция вычисляет, но производит еще и побочный эффект: меняет значение входного параметра так, что после выполнения функции оно становится равным единице. В результате после выполнения оператора M=FAKT(L)*L; получается значение M, равное L!, а не L!*L, что весьма неожиданно на первый взгляд. Правильная реализация этой функции такова:

```
FAKT: PROC (K) RETURNS(BIN FIXED);
DCL (K,F,I) BIN FIXED;
F=1; DO I=2 BY 1 TO K; F=F*K; END;
RETURN(F); END;
```

Хотя здесь имеется лишняя переменная, но этой функцией безопасно пользоваться.

Побочный эффект может проявиться и во внутренних процедурах в связи с возможностью пользоваться переменными из объемлющих блоков. Например, пусть в той же процедуре-функции FAKT забыли описать переменную I. Если эта процедура внешняя, то ничего не случится, ибо переменная I будет принята по умолчанию, но если эта процедура внутренняя и во внешнем блоке используется какая-то переменная I, то процедура FAKT ее будет портить. Например, в следующей ситуации:

```
PR:PROC OPTIONS(MAIN);
DCL I BIN FIXED, ...;
FAKT:PROC (K) RETURNS(BIN FIXED);
DCL (K,F) BIN FIXED;
F=1; DO I=2 BY 1 TO K; F=F*I; END;
RETURN(F); END;
…
DO I=1 TO 10;
   Z= .../FAKT(10);
END;
END;
```

цикл выполнится только один раз: после обращения к процедуре значение I окажется равным 11, в итоге цикл завершит работу, причем после выхода из него I будет иметь значение 12. Этот пример еще раз показывает, что привычка все описывать предохраняет от некоторых ошибок.

6.4. Передача параметров ссылкой, значением и через внешние переменные

Методы передачи параметров. В современных языках программирования обычно используются три способа передачи параметров подпрограммам. Первый способ - *передача значением* - заключается в том, что подпрограмме передается копия значения фактического параметра. Поскольку процедура работает только с копией, то доступа к фактическому параметру она не имеет, тем самым фактический параметр защищается от изменений. Однако снятие копии требует времени и памяти.

Второй способ - *передача ссылкой* - предполагает сообщение процедуре места в памяти, где размещается фактический параметр. Этот способ наиболее экономен с точки зрения затрат времени и памяти. Процедуре доверяется то место в памяти, куда помещен фактический параметр, и процедура может делать там все, что надо, а при ошибках также и то, что не надо. Понятно, что выходные параметры можно передавать только ссылкой, но не значением.

Третий способ - *связь через общие переменные* - заключается в том, что некоторая часть памяти не принадлежит ни одной из процедур, но ее расположение известно обеим процедурам. Одна процедура помещает туда какие-то данные, а другая - берет их оттуда. Так можно организовать связь между подпрограммами, если подпрограмме передаются всегда одни и те же фактические параметры, хотя, возможно, с разными значениями. Формально эти переменные не являются параметрами, но их можно условно считать таковыми, ибо они так же, как и параметры, служат для связи между подпрограммами.

Языковые средства ПЛ/1. В ПЛ/1 предусмотрены только два способа обмена информацией с процедурами: передача всех параметров *ссылкой*, а кроме того, возможна связь через *глобальные и внешние* переменные, фактически - это связь через общие переменные.

Однако имеются два нюанса, которые позволяют организовать в отдельных случаях некий аналог передачи по значению. По правилам ПЛ/1, если фактический параметр является выражением, отличным от идентификатора переменной, то формируется вспомогательная переменная без известного программисту идентификатора, куда помещается значение выражения, и в качестве фактического параметра выступает уже эта переменная. Таким образом, обращения к процедуре S

CALL S(A1); и CALL((A1));

отличающиеся только тем, что во втором обращении параметр взят в скобки, дадут разный результат: взятый в скобки идентификатор это уже выражение, отличное от идентификатора, следовательно, ЭВМ отведет под его значение вспомогательную переменную (хотя значение этого выражения и совпадает со значением A1) и именно эта вспомогательная переменная доверяется процедуре. В результате вычисления результат попадает во вспомогательную переменную, а A1 остается без изменения. Иногда этой особенностью можно воспользоваться: получаем как бы *передачу параметров значением*, так как процедуре доверяется не сама переменная, а копия ее значения. Этим достигается характерная для этого способа передачи параметров защита переменной - фактического параметра от изменения (порчи) ее процедурой, так как процедура просто «не знает», где находится сама переменная. Естественно, такая защита требует и времени и памяти, но иногда этот прием полезен: если процедура портит входной параметр, а его надо сохранить, то вместо снятия копии оператором присваивания достаточно заключить параметр в скобки.

Чаще же эффект передачи параметров значением возникает результате ошибки, связанной со вторым нюансом ПЛ/1. Если формальный и фактический параметры процедуры отличаются по типам, но эти типы допускают преобразование одного к другому, то также формируется вспомогательная переменная нужного типа, которой присваивается значение фактического параметра с соответствующим преобразованием типов. Процедуре передается именно эта вспомогательная переменная. Например, пусть имеются следующие описания и операторы:

```
DCL A FLOAT, B BIN FIXED,
F ENTRY (FLOAT),...;
... CALL(A); ...
... CALL(B); ...
```

Первое обращение выполнится абсолютно правильно. Что касается второго, то здесь параметр B имеет не тот тип, который нужен процедуре, поэтому

транслятором в программе будет заведен еще одна переменная без известного программисту идентификатора (мы его обозначим B') с атрибутами FLOAT, и второй оператор CALL преобразуется в B'=B; CALL(B'); Если процедура F только читает данные из своего параметра-массива, то никаких проблем не возникает, но если задачей процедуры F является выполнение в нем каких-то изменений, т. е. параметр выходной, то эти изменения будут произведены в B', а B сохранит прежние значения, что может немало удивить программиста.

Все сказанное выше в равной мере относится к функциям.

Связь по внешним и глобальным переменным. Внутренние процедуры имеют доступ к переменным, описанным в объемлющем блоке, если переменная с таким же идентификатором не описана заново в самой процедуре, это позволяет одной процедуре записать в глобальные переменные некоторую информацию, которой затем пользуется другая процедура. Поскольку в этом случае процедуры пользуются одним и тем же описанием глобальной переменной, то никаких неприятных эффектов возникнуть не может: обе процедуры знают тип глобальной переменной. Если же в двух внешних процедурных блоках описаны переменные с одинаковыми идентификаторами и с атрибутом EXTERNAL у каждой, то это предполагает, что этим переменным будет отведено одно и то же место в памяти, т. е. что это одна и та же переменная, которая не принадлежит ни одной из процедур. Однако при сопоставлении внешних переменных учитываются только их идентификаторы, и если другие атрибуты не совпадают, то получаются те же эффекты, что при несовпадении типов формальных и фактических параметров. Например, пусть имеются следующие процедуры:

```
GOL:PROC OPTIONS(MAIN);
DCL A BIN FIXED EXT, B FLOAT EXT, C CHAR(10) EXT;
A=1; CALL ALFA; PUT LIST(A); ...
C='1234567890'; CALL BETA (...);
END;
-------------------------
ALFA:PROC;
DCL A FLOAT EXT, B BIN FIXED;
... B=A;... A=20.5; ...
END;
-------------------------
BETA:PROC (X);
DCL X BIN FIXED, B FLOAT EXT, C CHAR(5) EXT;
…
END;
```

Здесь после обращения к процедуре ALFA при выполнении присваивания «B=A;» будет сделана попытка расшифровать целое значение как действительное и возникнут те же эффекты, которые рассматривались выше. Если процедура ALFA все же завершит работу, то после возврата в головную программу будет напечатано странное значение, получающееся от расшифровки действительного значения 20.5 как целого.

С переменной B никаких проблем не возникнет: так как в процедуре ALFA идентификатор B не имеет атрибута EXTERNAL, 'го процедура ALFA будет работать со своей локальной переменной B, не трогая внешнюю переменную B. Если более никаких процедур в программе нет и процедура BETA также не использует внешнюю переменную B, то атрибут EXT у переменной B из головной программы смысла не имеет, так как к ней никто более не имеет доступа.

Что касается внешней переменной C, то процедура BETA имеет «законный» доступ только к первым пяти символам C, откуда она может прочитать значение '12345' и куда может что-либо записать, и головная программа затем сможет это корректно обработать. Попытка обработать оставшуюся часть строки приведет к возникновению ситуации SUBRANGE –ERROR(19). Впрочем, если эта ситуация выключена, что имеет место по умолчанию, то процедуре BETA удастся обработать всю строку.

Глава 7. ОБРАБОТКА ПРЕРЫВАНИЙ

7.1. Ситуации и прерывания

Группы ситуаций. В соответствии с документацией в ПЛ/1 выделяется несколько групп ситуаций:

вычислительные - переполнение (OVERFLOW, OFL), целое переполнение, точнее, переполнение при работе с числами с фиксированной точкой (FIXEDOVERFLOW, FOFL), деление на нуль (ZERODIVIDE, ZDIV), исчезновение порядка (UNDERFLOW, UFL), преобразование (CONVERSION - ERROR(1)));

файловые (ввода-вывода) - конец файла (ENDFILE), не найден файл (UNDEFINEDFILE), ключ (КЕУ) и некоторые другие;

контролирующие - диапазон индекса (SUBSCRIPTRANGE- ERROR(16)), диапазон строки (STRINGRANGE – ERROR(19));

общие - ошибка (ERROR);

отладочные – контрольная точка (ERROR(35)).

Однако помимо этого в практике встречается еще одна группа ситуаций, которая не имеет наименований в терминах ПЛ/1, и в работах по ПЛ/1 не упоминается. Эту группу можно назвать машинные ситуации, ибо в основном они связаны с машинными ошибками, описанными в гл. 1.

Возможность выключения ситуаций. Контролирующие и отладочные ситуации - по умолчанию выключены, остальные ситуации - включены всегда и не выключаются.

Причины этого различия заключаются в том, чем контролируется возникновение ситуации.

Вычислительные ситуации контролируются аппаратурой машины (за исключением ситуации CONVERSION - ПЕРЕВОД, которая частично контролируется библиотечными подпрограммами ПЛ/1). Включенные вычислительные ситуации не тратят времени ЭВМ, так как аппаратура ЭВМ автоматически ведет соответствующий контроль, поэтому выключение этих ситуаций не ускоряет программу.

Включение контролирующих и отладочных ситуаций заключается во вставке транслятором соответствующих проверок в текст программы. Понятно, что включенная отладочная ситуация расходует заметное время на выполнение проверок своего возникновения и удлиняет программу. В хорошо отлаженной программе эти ситуации возникать не должны. Поэтому после отладки программы, если требуется высокая скорость выполнения, контролирующие и отладочные ситуации можно

выключить. Хотя, если скорость и объем памяти не критичны, их лучше оставить, так как в случае их возникновения из-за недостаточной отлаженности программы.

Легче разбираться с диагностическими сообщениями ситуации, чем с неопределенными эффектами.

Контроль возникновения файловых ситуаций выполняют библиотечные подпрограммы ПЛ/1, образующие так называемую файловую систему, и подпрограммы операционной системы. Во всех файловых ситуациях продолжать работу при их возникновении невозможно, поэтому они не выключаемы.

Пустая реакция на прерывание. Даже на не выключаемую ситуацию можно задать пустую реакцию, тогда при возникновении ситуации просто происходит переход в точку нормального возврата, т. е. внешний эффект такой же, как и при выключенной ситуации, но ЭВМ расходует больше времени, и программа становится длиннее (занимает больше памяти в ЭВМ). В подавляющем большинстве случаев попытка «закрыть глаза» на ситуацию, по которой надо предпринять какие-то действия, приводит к зацикливанию программы на возникновении этой ситуации и выполнении пустой ее обработки.

7.2. Задание реакции на прерывание

Операторы ON и REVERT. Задание реакции на прерывание выполняется оператором ON, имеющим вид

ON «имя ситуации» {«блок»};

Отметим, что в ON допустим только один оператор, причем составной недопустим. Поэтому при необходимости приходится использовать BEGIN-END, а не DO-END.

Оператор REVERT «имя ситуации»; отменяет действие оператора ON с соответствующей ситуацией. Если для этой ситуации операторов ON не было, то оператор REVERT ничего не изменяет.

Область действия оператора ON. При входе в каждый блок наряду с памятью под переменные ЭВМ отводит память под информацию о необходимой обработке прерываний. Для каждой ситуации отводится своя память, которую мы будем называть регистром ситуации. В начале работы главной процедуры ставится реакция аварийной выдачи и окончания на все ситуации, после которых нельзя продолжать работу. При выполнении оператора ON в регистр соответствующей ситуации заносится информация о необходимом действии, а предыдущая информация стирается. Оператор REVERT заносит в регистр исходную реакцию. В начале выполнения нового блока (обычного или процедурного) для него заводятся свои регистры ситуации, куда копируется значение из регистров ситуаций объемлющего

блока. Далее операторы ON заносят в них свою информацию, а операторы REVERT восстанавливают исходное значение регистров, повторно копируют в них содержимое регистров из объемлющего блока. По завершении блока или выхода из подпрограммы любым способом (по RETURN или GO TO) вся локальная память блока, в том числе регистры ситуаций блока, стирается, тем самым возобновляется действие оператора ON, работавшего в том блоке, в который вернулись.

В PL/1-KT ON-операторы восстанавливаются только при выходе из процедурного блока.

Таким образом, действие оператора ON распространяется до выполнения следующего оператора ON с той же ситуацией или оператора REVERT с той же ситуацией или до окончания работы блока.

Область действия ON определяется динамически, т.е. по выполнению оператора. Иными словами, действие ON распространяется на все операторы, которые выполняются после выполнения данного ON и до выполнения следующего ON, независимо от их взаимного расположения.

Отметим также, что при выполнении заданных в операторе ON действий все переменные берутся из того блока, где расположен оператор ON (для этого блока они могут быть как локальными, так и глобальными), а не из того блока, где произошло прерывание.

Например, если в программе

```
BEGIN; DCL A FLOAT; ...
ON OFL PUT LIST(A);
BEGIN; DCL A(3) FLOAT;
... Z=A(1)*A(2); ...
END;END;
```

случится переполнение в операторе присваивания, то будет напечатана переменная A из внешнего блока, а не массив A из внутреннего блока.

Примеры. При вычислении среднего арифметического по формуле $(a_1+a_2+...+a_n)/n$ результат будет всегда по модулю меньше максимального из модулей чисел, а потому пригоден для хранения в памяти ЭВМ. Но промежуточные значения могут оказаться очень большими, что вызовет переполнение. Чтобы избежать переполнения, надо переписать формулу в виде $(a_1/n+a_2/n+...+a_n/n)$ и программировать деление на n каждого элемента. Но такое решение замедляет программу, поэтому запрограммируем вычисление обычным образом, а если переполнение действительно произойдет, тогда перейдем на расчет по второй формуле:

```
ON OVERFLOW GOTO AV;
S=0;
DO I=1 TO N; S=S+A(I); END;
```

```
S=S/N; GO TO KON;
AV: S=S/N; DO I=I TO N; S=S+A(I)/N; END;
KON: ...
```

При переполнении происходит переход на метку AV (АВария), значение S при этом сохраняется, но i-й член еще не добавлен, поэтому продолжение суммирования идет с i-го члена и до конца.

Аналогия. *Иногда вызывает сомнение сохранность значения S, но ее можно объяснить следующим образом. Если вы взяли предмет в руки, чтобы им заменить другой предмет, а первый предмет рассыпался у вас в руках, следует ли из этого, то испорчен второй предмет? Если бы переполнение произошло при присваивании (чего не может быть), то к моменту переполнения возможно что-то было бы испорчено в S, но оно происходит при суммировании, когда до засылки в S нового значения дело еще не дошло.*

Правильно сработает и вариант с внесенным в цикл ON:

```
DO I=1 TO N;
  ON OVERFLOW GOTO AV;
S=S+A(I);
END;
```

но он будет работать значительно медленнее, так как ЭВМ на каждом проходе цикла будет повторно заносить в регистр ситуации информацию о реакции на переполнение.

Аналогия. *Если бы вы суммировали массив вручную, а вам бы все время говорили под руку: «Смотри, не забудь про переполнение», то вряд ли это способствовало бы ускорению счета.*

Интересной ошибкой является вариант

```
DO I=1 TO N;
S=S+A(I);
ON OVERFLOW GOTO AV;
END;
```

где действие оператора ON предполагается в варианте «если было переполнение...». Вообще говоря, действие ON на предшествующие операторы не распространяется, но в данном случае результат будет правильный. При первом проходе цикла переполнения быть не может, ибо начальное значение S нулевое и после первого сложения результат будет равен A(1), т. е. в памяти представим. Далее оператор ON сообщает ЭВМ о необходимой реакции на переполнение и при втором

проходе цикла ЭВМ будет иметь это в виду. Затем на каждом проходе цикла будет снова и снова перезаписываться регистр ситуации, но одной и той же информацией.

Рассмотрим несколько примеров, иллюстрирующих динамику действия оператора ON. Пусть имеется такой фрагмент программы:

```
ON <ситуация1> <реакция1>;
DO I=1 TO N;
... <оператор1> ...
IF ... THEN ON <ситуация1> <реакция2>;
... <оператор2> ...
END;
```

Пусть в операторе возникла ситуация, какая будет реакция: реакция1 или реакция2? Ответ таков. На первом проходе цикла будет реакция1, так как второй оператор ON еще не успел сработать. На последующих проходах эта реакция сохранится до тех пор, пока в IF не выполнится условие - тогда сработает второй ON и на последующих проходах цикла уже будет срабатывать реакция2. То же относится и к возникновению ситуации1 в операторе2 с тем лишь различием, что реакция2 будет работать не со следующего, а с того самого прохода цикла, на котором выполнилось условие.

Немного изменим фрагмент программы:

```
ON <ситуация1> <реакция1>;
DO I=1 TO N;
... <оператор1> ...
IF ... THEN ON <ситуация1> <реакция2>;
       ELSE ON <ситуация1> <реакция3>;
... <оператор2> ...
END;
```

Теперь при возникновении ситуации1 в операторе1 на первом проходе цикла также будет срабатывать реакция1, а на последующих проходах - либо реакция2, либо реакция3, в зависимости от того, выполнилось ли условие в IF на предыдущем проходе.

Для иллюстрации взаимодействия нескольких областей действия ON рассмотрим программу, которая вводит два массива по 100 элементов в каждом, выполняет какой-то расчет, затем в те же массивы вводит новые значения и выполняет этот же расчет для них и так далее, пока не исчерпаются данные. Такое построение программы достаточно типично для многих применений, оно позволяет выполнить расчет по произвольному числу вариантов исходных данных.

```
PR:PROC OPTIONS(MAIN);
DCL (A,B)(100)FLOAT. I BIN FIXED, OSH BIT(1), ...;
C:OSH='0'B; I=1;
ON ERROR(1)
```

```
BEGIN;
   OSH=’1’B;PUT EDIT('ОШ. A(',I,')')(SKIP,A,F(3),A);
END;
ON ENDFILE(SYSIN) GO TO KON;
GET LIST(A(1));
ON ENDFILE(SYSIN)
BEGIN;
      PUT EDIT('НЕ ХВАТАЕТ ДАННЫХ') (SKIP,A);
      GO TO KON;
END;
GET LIST ((A(I) DO I=2 TO 100));
ON ERROR(1)
BEGIN;
    OSH='1'B; PUT EDIT('ОШ. B(',I,')')(SKIP,A,F(3),A);
END;
GET LIST ((B(I) DO I=1 TO 100));
IF OSH THEN GO TO C;
<выполнение расчета>
GO TO C;
KON:END;
```

В этой программе обрабатываются две ситуации: CONVERSION-ERROR(1) и ENDFILE. Первый оператор ON ENDFILE(...) распространяет свою область действия только на первый оператор GET. Если при вводе первого элемента A возникла ситуация ENDFILE, то это значит, что данные просто закончились и надо завершать работу. Затем второй оператор ON ENDFILE(...) устанавливает другую реакцию на эту ситуацию, в его область действия попадают оба оставшихся оператора GET, теперь возникновение ситуации ENDFILE диагностируется как ошибка. После выполнения расчета для одних данных цикл начинает работу сначала, и опять первый ON ENDFILE устанавливает свою реакцию.

Независимо от работы операторов ON ENDFILE работают два оператора ON ERROR(1), которые также по очереди устанавливают свои реакции, в области действия первого оказываются операторы ввода A, в области действия второго - оператор ввода B.

Теперь предположим, что по каким-то причинам потребовалось максимально ускорить программу при обработке ошибочных данных, для чего сделано следующее. Когда выполняется доввод массива, то вводимые данные нам уже не нужны и никакой необходимости присваивать их элементам массива нет. Если учесть также, что индексация весьма трудоемкая операция, то имеет смысл вводить ненужные данные в какую-нибудь одну простую переменную: каждое следующее

число будет стирать предыдущее, в итоге в этой переменной останется последнее значение, что для нас роли не играет.

Использование функции ONCODE. При возникновении ситуаций системная переменная ONCODE принимает числовое значение типа BIN FIXED, которое соответствует возникшей ситуации, однако одной и той же ситуации соответствует несколько значений ONCODE, что позволяет уточнить причину возникновения ситуации. Но самым ценным является то, что ситуации, не имеющие названий в терминах ПЛ/1 - адресация, спецификация, операция и т. д. - также имеют свои коды, что позволяет задавать реакции и на них:

```
ON ERROR
BEGIN;
IF ONCODE=...
     THEN <обработка ситуации>;
     ELSE LEAVE ON;
END;
```

Глава 8. ФАЙЛЫ

8.1. Выбор типов файлов

Классификация файлов. Согласно документации файлы на ПЛ/1 классифицируются по нескольким признакам: по доступу (прямые и последовательные), по способу передачи (потоком и записями) и т. д. Эта классификация рассматривает файлы с точки зрения ЭВМ, но программисту для правильного выбора типа и прочих деталей организации файла нужно знать классификацию файлов, ориентированную на их использование. С этих позиций файлы можно классифицировать следующим образом.

По способу передачи файлы делятся на предназначенные для восприятия:

- *человеком* или для получения информации от человека, а потому содержащие информацию в символьном представлении;
- *машиной*, а потому содержащие информацию в машинном (внутреннем, кодированном) представлении.

По *назначению* файлы разделяются на вводные, выводные и обновляемые (это деление не требует комментариев).

По *доступу* файлы делятся на *прямые* и *последовательные* (что тоже не требует комментариев), а в рамках одного доступа различаются способом организации.

***Замечание**. Мы будем для краткости использовать термин «прямой файл», хотя формально правильным является термин «файл с прямым доступом», или «файл прямого доступа».*

Критерии выбора. Выбор *способа передачи* теперь становится понятным: передачу «потоком» целесообразно применять только для файлов, которые либо подготовлены человеком (набор текста на устройствах подготовки информации), либо предназначены для вывода результатов. Разумеется, их можно применять и для обмена информацией машина-машина, но это неэффективно, так как преобразование в символьное представление и обратно требует времени.

Выбор *назначения* однозначно определяется задачей. Выбор *метода доступа* также определяется задачей. Разумеется, всегда можно выбрать файл прямого доступа и обрабатывать его последовательно, но, выбрав прямой файл, программист неявно подключает к своей программе средства его обработки, а для прямых файлов они более сложные. Таким образом, выбор прямого файла там, где достаточно последовательного, приведет к потере эффективности (скорости) программы.

Аналогия. *Не стреляйте из пушки по воробьям - не экономно.*

Выбор *способа организации* также связан с теми действиями, которые будут проводиться с файлом. Эффективность того или иного способа организации будет рассмотрена ниже.

8.2. Расчет длины записей

Память, занимаемая переменными. Подсчет числа байтов, участвующих в обмене можно выполнить по табл. 1.2 из п. 1.4 со следующими изменениями. Память под простые переменные берется по таблице, при этом память под строки с атрибутом VARYING вычисляется по фактическому количеству символов, хранящихся в строке к моменту выполнения оператора WRITE плюс байт, где записана длина.

Память для массивов и структур. Объем памяти под массив вычисляется умножением числа элементов массива на объем памяти, занимаемый одним элементом. В структурах объем памяти подсчитывается суммированием объемов памяти, занимаемых элементарными данными.

Строки переменной длины в обмене записями. Из всего сказанного выше остается неясным, как подсчитывать память под запись для массивов из строк VARYING.

Реализация PL/1-KT предусматривает специальный режим чтения и записи строк переменной длины операторами READ и WRITE.

8.4. Работа с последовательными файлами

Пример. Для иллюстрации типичных приемов работы с последовательными файлами и некоторых ошибок в работе с ними рассмотрим программу слияния двух файлов с одинаковой структурой записей. Записи содержат строку из 10 символов и некоторую дополнительную информацию, записи в файлах упорядочены по лексикографическому возрастанию строк. Требуется слить информацию из двух исходных файлов в третий выходной так, чтобы в нем сохранилась упорядоченность строк.

Программа слияния имеет следующий вид:

```
SLIJAN:PROC OPTIONS(MAIN);
DCL (F1,F2,FREZ) FILE;
1 Z1, 2 ST CHAR(10), 2... ,
```

```
1 Z2, 2 ST CHAR(10, 2 ... ;

OPEN FILE(F1) INPUT RECORD SEQL;
OPEN FILE(F2) INPUT RECORD SEQL
OPEN FILE(FREZ) OUTPUT RECORD SEQL;
ON ENDFILE(F1) GOTO KF1;
ON ENDFILE(F2) GOTO KF2;
READ FILE(F1) INTO (Z1); READ FILE(F2) INTO (Z2);
C:IF Z1.ST < Z2.ST
   THEN DO; WRITE FILE(FREZ) FROM(Z1);
                READ FILE(F1) INTO(Z1); END;
   ELSE DO; WRITE FILE(FREZ) FROM(Z2);
                READ FILE(F2) INTO(Z2); END;
GO TO C;
KF1: ON ENDFILE(F2) GO TO K;
C1:WRITE FILE(FREZ) FROM(Z2);
    READ  FILE(F2) INTO(Z2); GO TO C1;
KF2: ON ENDFILE(F1) GO TO K;
C2:WRITE FILE(FREZ) FROM (Z1);
    READ FILE(F1) INTO(Z1); GO TO C2;
K: CLOSE FILE(F1), FILE(F2), FILE(FREZ);
END;
```

Отметим, что запись в файл FREZ идет из разных структур. В остальном программа построена как обычно: цикл с выходом из него по ENDFILE в одном из файлов, далее еще цикл, дописывающий оставшийся файл, выход из которого также идет по ENDFILE.

Эта программа совсем не оптимальна: когда один из файлов исчерпан и идет дописывание второго, исчерпанный файл остается не закрытым. Но открытый файл занимает оперативную память и может занимать внешнее устройство. Поэтому в строки с метками KF1 и KF2 имеет смысл добавить операторы CLOSE, закрывающие файлы F1 и F2 соответственно. При этом оператор CLOSE с меткой K можно оставить без изменений: повторное закрытие уже закрытого файла не влечет за собой никаких действий, и этот оператор закроет файл FREZ и тот из файлов F1 и F2, который еще не закрыт.

Об автоматическом открытии-закрытии файлов. Рассмотрим некоторые интересные ошибки в работе с файлами. Пусть после меток KF1 и KF2 забыли поставить операторы ON, меняющие обработку ситуации ENDFILE, тогда будут действовать операторы ON, стоящие в начале программы. Пусть раньше кончился файл F1 и цикл, начинающийся меткой C2, дописывает файл F2, пока он не кончится. По концу файла произойдет переход на метку KF2 и будет сделана попытка прочесть запись из F1: опять возникнет ENDFILE(F1) и программа зациклится между метками KF1 и KF2.

Если операторы ON с метками KF1 и KF2 заменить операторами CLOSE, закрывающими исчерпанный файл, то произойдет следующее. Например, пусть первым исчерпан F1. Он будет закрыт, произойдет дописывание записей из F2, по окончанию F2 произойдет переход на метку KF2, где F2 закрывается и читается закрытый файл F1. Здесь произойдет автоматическое открытие F1 и он будет читаться с начала (!). Программа будет работать, пока не исчерпает все место на носителе, отведенное файлу FREZ, после чего завершится, выдав сообщение операционной системы об ошибке. Пример иллюстрирует вредность автоматического открытия файлов.

Исчерпание места на носителе. При работе с выводными файлами возможен вывод такого количества записей, которое не умещаются в отведенное файлу место на диске или ленте. В этом случае возникает ситуация ENDFILE, которая первоначально была определена только для вводных файлов.

8.5. Работа с файлами прямого доступа

Общие возможности. Файлы прямого доступа отличаются от файлов последовательного доступа тем, что в них у каждой записи имеется некоторый уникальный признак, позволяющий идентифицировать эту запись. Такой признак называется ключом записи. При наличии такого признака можно в операторах обмена с этим файлом указывать его, тем самым определяя с какой записью файла будет идти работа.

В PL/1-KT предусмотрен только один вид файлов прямого доступа, где записи в файле просто пронумерованы. На внешнем носителе ключи не хранятся (нет необходимости). Файл занимает минимум места на носителе и обрабатывается быстро, так как позицию записи в файле можно легко вычислить.

8.10. Последовательный доступ к прямым файлам

Назначение. В файлах с прямым доступом записи располагаются в некотором порядке и в принципе ничто не мешает выбирать их не по ключу, а подряд одну за другой в том порядке, в котором они расположены в файле. Так можно создавать новый файл, читать и модифицировать файл, Для последовательной обработки прямого файла в его описании надо заменить атрибут DIRECT на SEQUENTIAL (или опустить этот атрибут вообще) и добавить атрибут KEYED.

Работа операторов ввода-вывода. Если файл выводной, то запись в него выполняется оператором

```
WRITE FILE (<имя>) FROM (<переменная)>)
                   KEYFROM (<выражение>);
```

В PL/1-KT вместо KEYFROM можно писать ключевое слово KEY.

Т.е. так же, как и при прямом доступе, однако ключ очередной записываемой записи должен быть больше, чем ключ последней уже записанной записи, в противном случае возникает ситуация KEY. При записи таким способом в файлы пропущенные записи автоматически заполняются пустыми (фиктивными записями). Например, если в файл после записи с ключом 22 заносится запись с ключом 25, то фактически сначала будут занесены две пустые записи с ключами 23 и 24, а затем запись с ключом 25.

Из вводного файла записи читаются операторами

```
READ FILE(<имя>) INTO(<переменная>)
                 [ KEYTO(<переменная>) ];
```

Каждый оператор READ считывает очередную запись; если задан режим KEYTO, то кроме считывания самой записи, в указанную после KEYTO переменную помещается ключ прочитанной записи. Эта переменная должна быть с атрибутом BIN FIXED(31). Из файлов записи считываются все записи, включая пустые.

8.11. Передача файлов подпрограммам

Область действия имен файлов. Имена файлов, описанные так, как это делалось выше в данной главе, являются внешними (т. е. автоматически получают атрибут EXT). Это означает, что два описания файлов из разных внешних подпрограмм или параллельных блоков, описывающие файлы с одинаковыми именами, относятся к одному и тому же файлу, поэтому такие описания не должны иметь противоречивых атрибутов. Это также означает, что имена файлов не должны совпадать с именами других переменных, объявленных с атрибутом EXT, возможно, даже в другой

подпрограмме или другом внешнем блоке. Возможность сделать имя файла локальным в ПЛ/1 есть, но выходит за рамки нашего рассмотрения.

Таким образом, файл может быть частично обработан в одной подпрограмме (например, открыт там), а частично - в другой (например, прочитан и закрыт).

Передача имен файлов через параметры. Файл, являющийся формальным параметром подпрограммы, должен быть описан с атрибутом FILE и без остальных атрибутов. Применяемые к этому файлу операторы не должны противоречить атрибутам файлов, передаваемых в качестве фактических параметров. Например, если имеется подпрограмма копирования последовательных файлов:

```
KOPIR:PROC(F1,F2);
DCL (F1,F2) FILE, ZAP CHAR(*);
OPEN FILE(F1) INPUT;
OPEN FILE(F2) OUTPUT;
ON ENDFILE(F1) GO TO KON;
C:READ FILE(F1) INTO(ZAP); WRITE FILE(F2) FROM(ZAP);
GO TO C;
KON:CLOSE FILE(F1),FILE(F2);
END;
```

то к ней нельзя обращаться с фактическими параметрами, являющимися файлами «поток» (операторы READ и WRITE противоречат атрибуту STREAM), второй фактический параметр не может быть прямым файлом (так как тогда оператор WRITE должен иметь режим KEY).

8.12. Буферизация записей

Буферизация. Для обработки записей в памяти ЭВМ отводится буфер, т. е. область памяти, способная вместить блок данных. Первый оператор READ переписывает с внешнего носителя в буфер весь блок, а затем пересылает первую запись в нужную переменную. Второй оператор READ уже не выполняет чтение с внешнего устройства, а только пересылает очередную запись из буфера в переменную, т. е. второй оператор READ выполняется значительно быстрее. Если в блоке k. записей, то каждый k-й оператор будет выполняться медленно, а остальные - быстро.

При записи все происходит в обратном порядке: первые (k-1) операторов WRITE только записывают информацию в буфер, а когда буфер заполняется k-м оператором WRITE, происходит собственно запись на внешний носитель.

Аналогия. *Мелко фасованный товар, например, консервные банки, обычно «блокируют» в коробки, скажем, по 20 банок. Со склада приносят целый «блок»*

(коробку) и помещают его в «буфер» (рабочее место продавца), далее продавец только выдает по банке из «буфера». «Внешнее устройство» (склад) не может выдавать банку, оно может выдавать только целый «блок». Отметим, что чем крупнее блок, тем реже приходится обращаться к складу, тем быстрее идет работа, но есть и ограничения: коробка должна поместиться на рабочем месте и быть доступной по весу для подъема одним человеком.

В Windows имеется еще более удобный механизм «отображения файлов на память», позволяющий обращаться к файлам как к обычным массивам в памяти. В PL/1-КТ этим механизмом можно воспользоваться, если при открытии файла задать размер буфера минус единицей и затем достать адрес этого буфера в указатель, например:

```
DCL F FILE, P PTR;
OPEN FILE(F) INPUT ENV(B(-1))
READ FILE(F) SET(P);
```

Таким образом, буферизация записей ускоряет работу, но расходует память ЭВМ под буфер. Конкретный коэффициент буферизации следует выбирать, исходя из доступного объема памяти (как оперативной, так и внешней) и учета факторов конкретной программы: требуется ли высокая скорость или важнее сэкономить оперативную память ЭВМ.

Аналогичная ситуация имеет место с файлами «поток». Фактически мало какие устройства способны принимать или выдавать по одному символу (например, устройство чтения с перфокарт не может прочесть полперфокарты - только всю перфокарту). Поэтому файлы «поток» на самом деле состоят из блоков символов, из которых ЭВМ выбирает нужное количество символов. Таким образом, выдаваемая по PUT информация фактически накапливается в буфере, а собственно печать происходит только тогда, когда буфер заполнен или же сработал элемент формата SKIP, показывающий, что в эту строку более ничего выводиться не будет. При зацикливании программы последняя строка может оказаться не напечатанной: незаполненный буфер не записывается на носитель.

Аналогия. *Если файл «запись» - аналог торговли фасованным товаром, то файл «поток» - аналог торговли, скажем, молоком в разлив. Естественно, молоко все равно фасовано: в 40-литровых бидонах или 1000-литровых цистернах, но покупатель этого может и не знать - продавец по исчерпании одного бидона переходит к другому без ведома покупателя, пока не наберет заказанное количество товара.*

Глава 9. ОБЗОР СРЕДСТВ ЯЗЫКА

9.1. Критический обзор языка ПЛ/1

Назначение обзора. Освоив программирование на ПЛ/1 в рамках рассмотренных средств, программист, естественно, захочет посмотреть, какие еще средства имеются в языке ПЛ/1 и что из них может оказаться ему полезным. Изучить возможности языка можно по одной из книг, приведенных в списке литературы, однако в этих работах описаны лишь сами возможности без рекомендаций по их применению. ПЛ/1 создавался около 40 лет назад и с современных позиций далеко не удовлетворяет требованиям, предъявляемым к хорошим языкам программирования. Своей конечной целью эти требования имеют повышение удобства написания и отладки, а ;также надежности программ, написанных на данном языке.

Именно с этих позиций можно сказать, что некоторые возможности языка просто не следует применять, так как они снижают надежность программы или удобство отладки. Таким образом, данный обзор призван служить путеводителем человеку, желающему расширить свои знания языка ПЛ/1. Он призван показать, какие возможности следует изучить в первую очередь, а чем надо пользоваться с осторожностью.

Причины исключения возможностей из рассмотрения. Если перечислить возможности, предоставляемые программисту тем или иным языком, то список возможностей ПЛ/1 будет самым длинным. Поэтому детально рассмотреть их все в рамках одной книги сложно, и в наше рассмотрение попали только самые употребительные возможности и те, которые следовало бы использовать более активно, чем это делается в настоящее время. Прочие возможности можно разделить на две группы. Первую группу составляют полезные средства языка, которые просто более сложны для понимания, реже используются или используются для относительно малого количества сложных задач.

Что же касается возможностей второй группы, то относительно них можно сформулировать следующее утверждение: если создать язык ПЛ/2, отличающийся от ПЛ/1 только тем, что в нем будут отсутствовать возможности из второй группы, то мы получим язык лучший, чем ПЛ/1. Именно так и сделан стандарт X3.74. Относительно любого средства языка можно сказать и следующее: существует пример, когда применение именно данного средства дает наилучшую по каким-то параметрам программу (самую короткую, самую простую, самую быструю), но для средств из второй группы такие примеры достаточно искусственны.

Есть обширная критика языка ПЛ/1, но она имеется специальной литературе, ориентированной на разработчиков языков. Недостатки языка ПЛ/1 не умаляют ценности умения программирования на нем, так как этот язык весьма распространен.

Кроме того, умение программировать на ПЛ/1 позволяет человеку очень быстро освоить программирование на любом другом процедурном языке: половина материала, приведенного в любом учебнике по ПЛ/1 может быть изложена применительно к языкам АЛГОЛ-60, ФОРТРАН, Паскаль и др. путем простой замены фрагментов программ на ПЛ/1 фрагментами на соответствующем языке почти без изменения текстовой части, а главы, касающиеся ввода-вывода и работы с файлами, потребуют незначительной переработки. Вместе с тем даже программисту-непрофессионалу полезно знать критику языка, чтобы легче перейти на новые, более современные языки по мере их внедрения.

Критерии качества языка и их удовлетворение в ПЛ/1. Для того, чтобы лучше разобраться в последующих рекомендациях по изучению или не изучению дополнительных возможностей ПЛ/1, рассмотрим, какие основные требования предъявляются к языку и насколько ПЛ/1 им удовлетворяет.

Полнота описания и канонизированность языка. В идеале должно быть полное каноническое описание языка, в котором определено, как должна выполняться любая конструкция, а каждая реализация языка обычно имеет дополнение вида «по сравнению со стандартом в данной реализации нет того-то и/или имеется дополнительно то-то». У ПЛ/1 такого описания нет, и иногда единственный способ узнать, как должна сработать определенная конструкция, - это выполнить ее на ЭВМ: что ЭВМ сделает, то и правильно. Поэтому в ряде книг по ПЛ/1 можно встретить фразу «все сомнительные конструкции были проверены на ЭВМ в ОС ЕС версия ...» (указание версии существенно, так как разные версии могут дать разные результаты).Смена версии транслятора может привести к появлению ошибок в отлаженной программе, причем в некоторых случаях программа будет работать не так, как раньше, а в некоторых могут появиться даже синтаксические ошибки.

Канонизированность у языка есть и теперь она доступна через описание стандарта в Интернете

Машинная независимость. Язык должен быть определен так, чтобы его можно было реализовать на ЭВМ разных типов. Однако во многих конструкциях ПЛ/1 столь явственно проглядывают особенности ЕС ЭВМ (1ВМ/360), что реализация ПЛ/1 на других типах ЭВМ весьма затруднительна: вряд ли удастся насчитать более пяти типов архитектуры ЭВМ, на которых имеется транслятор с ПЛ/1. То есть ПЛ/1 - это язык почти для одного типа ЭВМ, хотя этот тип и очень распространен. Отдельные конструкции языка зависят даже от конкретной реализации.

Это сомнительное утверждение, так было сделано в ФОРТАНЕ, а не в ПЛ/1. Имеются трансляторы для самой распространенной архитектуры процессоров x86.

Синтаксическое единообразие. Этот вопрос решен в ПЛ/1 достаточно плохо: сходные конструкции имеют разный вид (условие после IF ограничивается пробелами, а после WHILE - скобками; заполнение массива константой с помощью атрибута INIT не похоже на то же с помощью оператора присваивания), а одинаковые конструкции могут начать совершенно разные вещи («A=B» может быть сравнением и присваиванием; «A (B)» может быть взятием элемента массива, а может быть обращением к функции и т.д.).

Естественность. Этот критерий строго определить невозможно, но в целом смысл его в том, что конструкция языка не должна противоречить здравому смыслу, а это в ПЛ/1 имеет место далеко не всегда (примером является взаимное преобразование строки в число и наоборот по правилам автоматических преобразований типов).

Преемственность. Это не критерий качества самого языка, но авторы языка должны помнить, что существуют и другие языки и имеется контингент программистов, перешедших к программированию на ПЛ/1 после работы на других языках. Поэтому желательно, чтобы конструкция, похожая на конструкцию другого языка, и работала так же. Заимствованные из КОБОЛа цифровые шаблоны означают в ПЛ/1 совсем не то, что в КОБОЛе, да и выглядят похоже, но не так же. Это снижает надежность программирования.

Минимальность и ортогональность. Основных понятий должно быть не много и они не должны дублировать друг друга. У программиста должно быть минимум вариантов скомпоновать из основных понятий то, что ему нужно. ПЛ/1 содержит очень много дублирующих друг друга средств (оставшихся в основном вне рассмотрения).

Расслоенность (концентричность). Человек должен иметь возможность изучить не весь язык, а только нужный ему слой (концентр) и работать в нем, даже не подозревая о существовании в языке чего-то еще. Хотя в большинстве книг по ПЛ/1 и говорится что это свойство в языке есть (научный работник может изучить так называемое ФОРТРАНное подмножество ПЛ/1, а коммерческий программист - КОБОЛьское подмножество), но на самом деле это не совсем так: концентры получаются незамкнутыми. Например, научный работник, которому не надо использовать переменные с атрибутами DEC FIXED, вынужден будет познакомиться с ними, так как без этого не понятно, как передаются через параметры числовые константы. Нельзя было не упомянуть и о принципе умолчания - иначе не ясна реакция ЭВМ на некоторые ошибки. По-видимому, все же некоторые концентры выделить можно, но они оказываются неудобными для применения. С

концентричностью языка связана и концентричность реализации: работая в рамках одного концентра, человек не должен получать от ЭВМ диагностическое сообщение с терминами из более крупного концентра. Этого непросто добиться в любом языке. Например, после изучения основных управляющих конструкций уже можно писать простые программы. Но при ошибке в описании массива вы можете получить от ЭВМ сообщение «обращение к функции в левой части оператора присваивания», не понятное в рамках минимального концентра.

Структурированность и модульность. Язык должен позволять писать программы, состоящие из отдельных и независимых модулей, - это ПЛ/1 позволяет. Язык должен иметь полный набор структурных конструкций - это в ПЛ/1 есть. С этой стороны ПЛ/1 заслуживает высокой оценки. Отметим мощную и удобную конструкцию цикла. Не совсем удачна конструкция ветвления: если бы для IF имелась некая «закрывающая скобка», как для DO имеется END, то это бы сняло возможность некоторых ошибок, рассмотренных в гл. 2.

Контролируемость. Конструкции языка и его реализация должны быть таковы, чтобы максимальное количество ошибок выявлялось в процессе трансляции или в процессе выполнения и выдавалось в виде диагностических сообщений, а не в виде неопределенных эффектов выполнения программы. Но ряд элементов языка ПЛ/1 (принцип умолчания, автоматическое преобразование типов и др.) дают противоположный эффект. В этой части ПЛ/1 уступает практически всем языкам.

Реализуемость и эффективность. Желательно, чтобы программа допускала быструю трансляцию и возможность быстрого выполнения. Но ряд конструкций языка ПЛ/1 допускает эффективную реализацию только на ЕС ЭВМ. Изобилие возможностей языка существенно замедляет трансляцию, и по той же причине выполняйся программа медленней, чем аналогичная программа, написанная на языке с меньшим количеством возможностей.

Аналогия. *Зерновой комбайн включает косилку, молотилку и веялку и эффективен при прямом комбайнировании. Но если его использовать сначала для косьбы в валки, а затем для обмолота, то работает треть комбайна, но горючее тратится на перемещение всего агрегата. Так и в ПЛ/1: редко когда язык используется на всю свою мощь, но «везет» на себе ЭВМ весь «агрегат» языковых средств.*

Тоже спорное утверждение. Не используемые возможности обычно никак не влияют на эффективность. Непонятно, почему только ЕС ЭВМ эффективна для

ПЛ/1? Например, реализация на x86 достаточно эффективна и хорошо укладывается в основные языковые конструкции.

Несмотря на существенное количество недостатков, ПЛ/1 успешно и широко используется в практике. Кстати, отметим, что языка программирования без недостатков пока еще не создали.

Рекомендации по использованию или не использованию тех или иных средств языка будут основываться на рассмотрении в основном надежности и эффективности получающейся программы.

9.2. Арифметические возможности

Комплексные числа. Имеется возможность описать, что переменная будет принимать комплексные значения; имеются и способы изображения комплексных чисел. Операции над комплексными значениями выполняются по правилам комплексной арифметики. Использование комплексных чисел часто необходимо в научных расчетах. Эта возможность, безусловно, полезна. Комплексные числа не реализованы в стандарте X3.74 и PL/1-KT.

Управление точностью. На самом деле в ПЛ/1 нет понятий «целое число» и «действительное число», а есть 1474 варианта описания числовой переменной (из которых практически полезны не более 100), отличающихся формой представления, основанием системы счисления и точностью. Программист может указать, что число представляется в двоичной (BINARY) или десятичной (DECIMAL) системе счисления, имеет фиксированную (FIXED) или плавающую (FLOAT) точку и задать хранимое число двоичных или десятичных цифр до и после точки. Что касается системы счисления, то программисту совершенно безразлично, в какой системе счисления работает ЭВМ, поэтому этот вопрос можно оставить на усмотрение ЭВМ. Вопрос выбора фиксированной или плавающей точки - это вопрос выбора вида погрешности: фиксированная точка задает абсолютную погрешность, плавающая точка - относительную. Задание точности до одного знака также никому не нужно. Во-первых, это нереализуемо на ЭВМ (строго говоря, припер с циклом из п. 5.2 должен был дать именно приведенные в примере результаты, если все работающие в примере переменные описать с атрибутом FLOAT (4), т. е. удерживать в расчетах ровно четыре знака, но ЭВМ все равно будет удерживать шесть знаков ибо ей так удобнее), во-вторых, точный подсчет необходимой погрешности крайне затруднителен и лучше удержать несколько лишних знаков. Если добавить к этому грубую ошибку авторов языка, в результате которой произведение двух чисел с атрибутами FLOAT(6) дает число с атрибутом FLOAT(6), а не FLOAT(12), а в

результате умножения 7E0*7E0 получаем 4E1 (т. е. 40), а не 49, то становится ясно, что точностью чисел с плавающей точкой лучше не управлять.

Для чисел с фиксированной точкой задание точности имеет больший смысл: например, при экономических расчетах надо считать с точностью до копейки, т. е. держать два знака после точки.

Реализация ПЛ/1 такова, что числа BIN FLOAT и DEC FLOAT реализованы одинаково (поэтому можно использовать BIN FLOAT или просто FLOAT), а числа BIN FIXED и DEC FIXED - по-разному. При этом числа DEC FIXED обрабатываются в арифметических операциях примерно в 6 раз медленнее и памяти занимают, как правило, больше. Их достоинства - отсутствие перевода в двоичное представление при вводе и обратного перевода при выводе. Поэтому их рекомендуют использовать только в экономических расчетах, где много ввода-вывода и мало вычислений. Переменные с атрибутом BIN FIXED и с ненулевым количеством цифр в дробной части практически применения не находят и недопустимы в стандарте X3.74

Выводы таковы. Для приближенных чисел берем атрибут FLOAT, если не хватает одинарной точности берем двойную. Сформулировать строгое правило, когда следует использовать DEC FIXED затруднительно. В крайнем случае можно обойтись и без них: если надо иметь какой-то знак точным, берем числа с нужным масштабом, так чтобы они стали целыми и описываем как BIN FIXED.

Аналогия. *Если имеется фасованный по килограмму сахар, а нам надо 700 грамм, мы берем фасовку, если надо 1350 г - берем две фасовки. Попытка взять ровно столько, сколько надо, замедлит работу и нам и продавцу.*

Вывод по шаблону. В языке имеется очень мощное средство преобразования чисел при выводе – шаблоны. С их помощью можно выдавать числа на печать практически в любом мыслимом виде, очень часто это самый удобный способ подавить ненужные ведущие или конечные нули, поставить запятую в числе вместо точки и т.п.

9.3. Средства сокращения записи

Принцип умолчания. Этот принцип весьма широко развит в языке ПЛ/1 и проявляется разным образом, причем некоторые его проявления полезны, хотя в большинстве своем они вредны, так как снижают надежность программы. Он призван облегчить написание программы и в целом заключается в том, что если о каком-то объекте программы ничего не сказано, то информация нем определяется автоматически по определенным правилам. На первый взгляд это очень удобно,

однако программисты уже давно поняли, что надо обеспечивать удобство чтения и надежность программы, а не удобство написания программы.

Первое проявление этого принципа - возможность не описывать переменные. Это - вредная возможность языка, так как она выключает часть синтаксического контроля, а значит, снижает надежность программы.

Одной из главных целей ввода нового стандарта ПЛ/1 была как раз необходимость запретить использование неописанных переменных.

Если вы решили не пользоваться принципом умолчания, не спасает дела: он начинает работать при ошибках написания или набора текста, но все же привычка все описывать снижает вероятность ошибки из-за автоматической подстановки каких-то атрибутов.

Второе проявление этого принципа - возможность опускать часть описания, которая берется стандартной (вне зависимости от других частей описания): например, если не указан атрибут EXT, то переменная считается не внешней, а локальной, хотя есть и специальный атрибут для задания локальности. Эту возможность, по-видимому, нельзя считать вредной, ее аналоги можно найти во многих языках.

Третье проявление принципа умолчания - так называемое *контекстуальное описание*: атрибуты приписываются идентификатору по тому контексту, в котором он употреблен. Например, если файл имеет атрибут PRINT, то можно не указывать атрибуты STREAM OUTPUT: они *выводятся* из атрибута PRINT. Так как атрибуты выводятся из употребления переменной, то ошибку такое умолчание повлечь не может и вредным его назвать нельзя.

Автоматическое преобразование типов. В ПЛ/1 имеется 172 варианта преобразования типов: значение почти любого типа может быть преобразовано почти к любому другому типу. В совокупности с принципом умолчания эта возможность языка выключает значительную часть синтаксического контроля. С современных позиций эта возможность считается вредной, тем более что правила преобразований достаточно сложны и не всегда очевидны. Например, если переменной C с атрибутом CHAR(1) хотели присвоить символ '5', но по ошибке потеряли апострофы, то оператор «C=5;» вместо сообщения об ошибке присвоит значение из одного пробела (!?). Если переменной P с атрибутом BIT(8) присвоить числовое значение 2 или -2.86, то в обоих случаях строка P получит значение '00100000'B (сложно догадаться, почему), если же теперь переменную P использовать в арифметических расчетах, то ее значение будет интерпретировано как число 32. Вывод однозначен: пользоваться этими возможностями не рекомендуется; если нужны какие-то преобразования, то надо пользоваться

имеющимися в ПЛ/1 явными средствами преобразования, что позволит избежать ошибок (например, если надо преобразовать число в строку, то надо пользоваться оператором PUT STRING). Однако совсем забывать об этих преобразованиях типов нельзя, так как ЭВМ выполняет их иногда помимо воли программиста.

Вынесенные форматы. Если в нескольких форматных выражениях имеются совпадающие части (в частности, совпадает весь формат или несколько совпадающих частей в одном форматном выражении), то такой формат можно записать в отдельном операторе специального вида и ссылаться на него. Возможность полезна, хотя и применяется не часто.

9.4. Управление памятью

Управляемые переменные. Наряду с автоматическим отведением памяти при входе в блок и освобождением ее при выходе из блока и статическим отведением, когда память отводится при начале выполнения программы (так она отводится под переменные с атрибутом EXT), есть возможность описать переменные так, что память под них будет отводиться только при выполнении специального оператора в программе и освобождаться при выполнении другого специального оператора. Эта возможность весьма полезна при программировании сложных информационных структур, которые по мере выполнения программы должны сильно изменяться по размерам, составу элементов и т.д. Но необходимо помнить, что такое принудительное выделение-освобождение памяти весьма трудоемко для ЭВМ, поэтому пользование этой возможностью в тех случаях, когда можно использовать более простые возможности (в рассмотренных выше программах, например) существенно замедляет программы.

Наложенные переменные. Можно описать переменные так, чтобы они занимали то же место в памяти, что и другие переменные («наложились» на другие переменные). Эта возможность позволяет экономить память, а также выполнять некоторые трюки, которые иногда бывают полезны. Если одна переменная наложена на другую, то всякое изменение одной из них автоматически влечет за собой изменение и другой.

Аналогия. *Если представить переменную в виде коробки с наклеенным ярлыком-идентификатором, то наложенная переменная - это коробка, на которую наклеены два ярлыка*.

Если в одной части программы нам нужен массив A, а в другой массив A не нужен, но нужен массив B, то для экономии памяти можно наложить массив B на

массив A, т. е. использовать для этих массивов одно и то же место в памяти. Пожалуй, наиболее полезно применение этой возможности для наложения символьного массива на строку: описания

```
DCL ST          CHAR(80),
    MST(1:80)CHAR(1) DEFINED(ST);
```

объявляют, что 80 символов с одной стороны рассматриваются как 80-символьная строка, а с другой стороны - как массив из 80 односимвольных строк. В результате выражение SUBSTR(ST,1,5) означает пятисимвольный отрезок строки, а выражение SUBSTR(ST,K,1), означающее K-й символ строки может быть во всех контекстах заменено на MST(K), т. е. на обозначение K-го элемента массива. С помощью наложений можно дать специальное имя, например строке матрицы; возможны еще более сложные наложения. Что же касается трюков, то они в современной теории программирования не поощряются.

Базированные переменные и указатели. Базированная переменная не имеет своего места в памяти, она помещается там, куда указывает связанная с ней переменная типа указатель (типа POINTER). Эта возможность полезна в ряде случаев, но достаточно опасна, так как никакого контроля за размещением базированных переменных ЭВМ не ведет и с помощью базированных переменных можно выполнять самые разнообразные трюки. При ошибках в работе с базированными переменными возникают весьма неожиданные эффекты. Базированные переменные дают возможность выполнять произвольные наложения переменных. Но при умелом использовании они позволяют эффективно реализовать сложные динамические структуры данных, эффективно обрабатывать файлы и т. д. Разумеется, их рекомендуется применять только тогда, когда обойтись другими средствами сложно, невозможно или невыгодно.

Использование внутреннего представления данных в памяти. Эта возможность языка ПЛ/1 в том виде, в котором она имеется, уникальна для языков высокого уровня. Она позволяет работать с внутренним кодом данных в виде последовательностей битов: функция UNSPEC позволяет прочитать значение переменной как последовательность битов или записать в переменную любую последовательность битов, в частности, записать такое значение, какое в этой переменной храниться не может. Естественно, что эта возможность вредна, так как она позволяет «взломать» защиту, которую обычно обеспечивают языки высокого уровня. Для использования функции UNSPEC необходимо знать применяемый способ кодирования для используемых в программе типов данных, хотя обычно языки высокого уровня стараются избавить программиста от необходимости таких знаний. Кроме того, внутренний код – это внутреннее дело ЭВМ (или транслятора) и даже разные трансляторы на одном и том же типе ЭВМ используют разные методы

кодирования, поэтому, как только в программе появляется функция UNSPEC, программа сразу же теряет свою машинную независимость, и даже на другом трансляторе может дать иной результат. Таким образом, использование этой возможности – всегда трюк, но иногда он оказывается необходим, например, чтобы расшифровать результаты, полученные с помощью других трюков.

Иногда использование внутреннего представления данных может существенно повысить эффективность программы, но это происходит за счет потери ее машинной независимости.

9.5. Прочие возможности

Рекурсивные процедуры. Некоторые алгоритмы могут быть описаны на языке ПЛ/1 значительно короче, если разрешить рекурсию, т. е. обращение процедуры (функции) к самой себе. Специальное описание подпрограммы может разрешить такое обращение, но для использования этого средства надо хорошо понимать его функционирование, так как короткое описание на ПЛ/1 может вылиться в очень долго работающую программу.

Асинхронное выполнение. Современные вычислительные системы обладают возможностью одновременно выполнять несколько программ (иногда система имеет несколько процессоров, каждый из которых выполняет программу независимо от других, иногда система только имитирует одновременное выполнение нескольких программ). В стандарте X3.74 такие средства не предусмотрены, их предоставляет сама операционная среда Windows.

Эти средства позволяют ускорить программу. Они очень полезны в задачах моделирования сложных систем и в других применениях.

Редактирование вывода. Во многих применениях, в особенности, в экономических задачах, требуется специальное оформление вывода (составление таблиц - это простейший пример). В ПЛ/1 имеется богатейший набор шаблонов, позволяющий выполнить простым способом довольно сложное редактирование выводимой информации. Есть возможность разбиения выводимой информации на страницы любого размера с оформлением заголовков, нумерации и т. д.

Файловые возможности. В данной книге рассмотрены только простейшие файловые возможности языка, при этом эффективность обрабатывающей файлы программы рассматривалась минимально. Реально программист может использовать гораздо больше возможностей и существенно улучшить эффективность программы, используя специальные средства Windows.

Препроцессорные средства. Эти средства представляют собой надстройку над языком, которая позволяет в процессе трансляции видоизменить текст программы, включив в него стандартные фрагменты, или даже создать новый текст, выбрать один из возможных вариантов текста, выполнить статическую настройку и т.д. Эти средства позволяют написать сложную программу, рассчитанную на все возможные случаи, а затем при трансляции получить эффективный частный случай программы, настроенный на особенности конкретной задачи. Для описания этих средств требуется отдельная глава. Они полезны при написании сложных программ.

Выравнивание данных. В PL/1-KT имеется возможность включить выравнивание данных (в основном это важно для данных типа FLOAT), что не требует архитектура процессора x86, но позволяет ускорить программу за счет более эффективного чтения данных процессором из памяти.

Ввод-вывод управляемый данными. Это средство позволяет выводить информацию с автоматическим снабжением ее идентифицирующей частью в виде идентификаторов выведенных переменных: «A=... B=…». Это очень удобное средство для отладочных печатей. На вводе информация записывается в аналогичном виде: значение вводится в ту переменную, идентификатор которой записан перед ним.

Список литературы

1. Безбородов Ю.М. От ФОРТРАНа - к PL/1: Основы языка PL/1
М.: Наука, 1984. 208с.

Введение в ПЛ/1 для тех, кто знает ФОРТРАН.

2. Гребенников Л.К., Лебедев В.Н. Решение задач на ПЛ/1 в ЕС.
М.: финансы и статистика, 1981. 302 с.

Довольно полное описание реализованной в ОС ЕС версии ПЛ/1 и необходимых средств операционной системы. По основами программирования содержательных примеров нет. Очень полно и подробно, с большим количеством содержательных примеров даны файловые возможности языка.

3. Малютин Э.А., Малютина Л.В. ПЛ/1 для начинающих.
М.: Финансы и статистика, 1985. 208 с.

Ясным языком с большим количеством красивых рисунков и аналогий кратко рассмотрены основные понятия языка ПЛ/1, этапы решения задачи на ЭВМ и небольшие содержательные примеры. Предназначена для первоначального ознакомления с программированием и с ПЛ/1 без деталей.

4. Программирование на ПЛ/1 ОС ЕС /М.И. Аугустон, Р.П. Балодис и др
М.: Статистика, 1985. 269 с.

Описано некоторое подмножество ПЛ/1 ОС ЕС и необходимые сведения по ОС ЕС и транслятору без содержательных примеров. Предназначена для начального знакомства с реализацией ПЛ/1 в ОС ЕС для умеющих программировать.

5. Программирование, отладка и решение задач на ЭВМ единой серии. Язык ПЛ/1 под ред. И.А. Кудряшова. Л.: Энергоатомиздат, 1989. 279 с.

Отличный справочник по языку ПЛ/1 и соответствующим элементам ОС. Для изучения ПЛ/1 не предназначена. Хорошо описаны выдачи и сообщения транслятора, редактора связей и сообщения времени выполнения.

6. Скотт Р., Сондак Н. ПЛ/1 для программистов.
М.: Статистика, 1977. 223 с.

Описано небольшое подмножество языка для первоначального знакомства с языком и с программированием.

7. Фролов Г.Д., Олюнин В.Ю. Практический курс программирования на языке ПЛ/1. М.: Наука, 1983. 384 с.

Наиболее полное описание версии ПЛ/1-F и ПЛ/1-О в ОС ЕС (недостаточно подробно описаны файловые возможности) и необходимые сведения по ОС ЕС. Предназначено как справочное пособие для людей, владеющих программированием.

8. Штернберг Л. Ф. Минимальный ПЛ/1. Основы языка, техника программирования. М.: Наука, 1992. 206 с.

Учебник по языку для начинающих. Описано минимальное подмножество языка, достаточное для работы в наиболее часто встречающихся случаях. Рассмотрено большое число примеров с анализом эффективности программ и наиболее типичных ошибок.