Oldies… But Goldies!

ПРАКТИЧЕСКИЙ КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ PL/1

по следам Г.Д. Фролова и В.Ю. Олюнина

Современное прочтение классического учебного пособия по программированию на языке PL/1

Королев 2021

Г. Д. ФРОЛОВ, В. Ю. ОЛЮНИН

ПРАКТИЧЕСКИЙ КУРС ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ PL/1

ИЗДАНИЕ ВТОРОЕ, СТЕРЕОТИПНОЕ

Допущено Министерством
высшего и среднего специального образования СССР
в качестве учебного пособия
для студентов высших технических
учебных заведений

МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ

1 9 8 7

Почти 40 лет назад в Советском союзе вышло учебное пособие Г.Д. Фролова и В.Ю. Олюнина, которое, вероятно, стало самым массовым учебником по языку программирования в нашей стране, поскольку общий тираж всех его изданий приблизился к 300 000 экземпляров.

С современной точки зрения эта книга бесполезна и безнадежно устарела. Например, в разделе 14.1. описывается связь с «консолью ЭВМ», что выглядело устаревшим еще во времена написания этой книги. Встречаются и курьезы. Так, в разделе 3.2. упоминается, что размерность массива в компиляторе PL/1(F) может достигать 63. Но даже, если элемент такого массива занимает лишь байт, а диапазоны всех индексов не больше 2, то в памяти этот массив займет 2^{63} байт, что невозможно и сейчас.

Вина ли авторов в этом? Разумеется, нет. Они проделали огромную работу, чтобы снабдить будущую армию советских программистов (поэтому такой и тираж) отличными справочными и учебными данными, причем сделали это в энциклопедическом стиле. Проблема была в том, что пришлось описывать чужую систему, доставшуюся «бесплатно», но потребовавшую считаться со всеми своими особенностями, которые никто не собирался улучшать или исправлять. И даже создание своего, отечественного компилятора с PL/1 для «Эльбруса» ничего не изменило – «Эльбрус» был недоступен подавляющему числу советских программистов, а доступны были только ЕС ЭВМ и только имеющиеся там компиляторы. И по мере схода со сцены ЕС ЭВМ уходил и PL/1, а данная книга стала не нужна (тем более студентам) и превратилась в памятник истории ИТ.

Так сложилось, что большую часть своей жизни, с 1987 года, я активно использую язык PL/1 и даже сопровождаю и развиваю компилятор с этого языка. Но при этом я никогда не работал на ЕС ЭВМ и не имел дела с какими-нибудь DD-операторами или индексно-последовательными файлами. «Мой» компилятор – это потомок компилятора, разработанного американским специалистом Гарри Килдэлом в начале 80-х и изначально предназначавшегося для персональных компьютеров.

Применение, сопровождение и развитие компилятора потребовало поиска и анализа имеющейся и довольно богатой литературы по PL/1. Конечно, важное место заняла и книга Фролова и Олюнина, которая подкупила меня своей математической строгостью и всеохватностью. Захотелось сохранить этот труд для возможного использования в будущем. Но для этого мало оцифровать книгу (что, кстати, уже не раз выполнялось). От этого книга не сделается ни полезной, ни современной. Требуется внимательно прочитать весь текст и изменить/убрать то, что действительно устарело, то, что в моем компиляторе работает не так, и добавить то, чего не было 40 лет назад.

Попробую сразу ответить на ряд недоуменных вопросов.

Вопрос: зачем вообще сохранять, тем более изменять учебное пособие по языку, который уже никогда не будет востребован?

Ответ: я считаю, что этот язык вполне может быть востребован. И даже не тот язык IBM, а тот, что впоследствии станет называться стандартом ISO/IEC 6522:1992. Сейчас бы это назвали новым языком. Тем более что конкретная его версия рассчитана на русскоговорящую аудиторию, а развитие давно не зависит от Запада. Недостаточное, но необходимое условие востребованности – наличие ясной документации, причем не старой, увековечивающей все особенности PL/1 от IBM с пакетной обработкой, а относящейся к современной версии языка и к современной среде. Сам по себе PL/1 очень и очень неплох для ряда предметных областей, весь мой успешный многолетний опыт это подтверждает.

Вопрос: руководство или учебник имеют смысл только для доступных средств. Кроме ничтожного числа людей, по-прежнему работающих на «мэйнфреймах» или их эмуляторах, PL/1 ведь сейчас большинству не доступен?

Ответ: сейчас стало доступно многое из того, что было недоступно ранее. Например, даже «мэйнфреймовский» компилятор IBM, который, наконец, перенесен в среду Windows. Но я, конечно, имею в виду свой компилятор, который бесплатен и доступен на сайте pl1.su. В целом он соответствует стандарту ISO/IEC 6522:1992. Но стандарт не должен быть догмой, иначе он консервирует все недостатки языка и со временем начинает не соответствовать современным процессорам и операционным системам. Поэтому предлагаемый язык и компилятор расширены, по сравнению со стандартом, а этот стандарт назывался подмножеством языка PL/1.

Вопрос: наверное, единственная практическая задача – это попробовать запускать имеющиеся у кого-нибудь древние программы на PL/1. Но ведь для другого транслятора придется изменять их тексты?

Ответ: нет, задача запускать программы 40-летней давности, не меняя в них ни буквы никогда и не стояла. Современный стандарт учитывает многолетний опыт использования PL/1. Важен не тот факт, что если кто-то имеет старое ПО и хочет его использовать, то тексты программ, рассчитанных на ЕС ЭВМ, придется менять, а то, что именно и сколько в них придется менять. Например, уже давно отказались от использования не описанных переменных, и все переменные теперь требуется явно описать. Но, на мой взгляд, тексты программ от таких исправлений только выиграют.

Вопрос: зачем брать в качестве основы какое-то древнее пособие для студентов, когда есть более современная (2008) документация, например, книга ведущего европейского специалиста Эберхарта Штурма, да еще и с многообещающим названием «The new PL/I for PC, Workstations and Mainframe»?

Ответ: этот уважаемый специалист описывает язык с некоторыми доработками и обновлениями, но все равно это та самая система IBM времен «пакетной» обработки программ. А разработка для персональных

компьютеров изначально не тянула за собой хвост технологий 60-х, предполагала реализацию подмножества языка в виде международного стандарта и интерактивный стиль общения с компьютером.

Что же касается собственно текста, то я предпочитаю классический труд на родном языке с примерами и вопросами. На мой взгляд, вопросы – важная и очень полезная часть пособия. Конечно, не предполагается, что читатель будет выполнять все упражнения, приведенные у Фролова и Олюнина. А вот вопросы типа «найдите ошибку в записи оператора», на мой взгляд, удачный прием. Я хотел бы понизить исходные разделы «Вопросы и упражнения» до рубрики «Проверь себя», с тем, чтобы все задания выполнялись за секунды или за минуту и не очень замедляли собственно чтение.

Вопрос: для компилятора с pl1.su уже есть документация. Зачем же составлять еще одну, да еще в форме учебного пособия? Ведь написание книги подобной Фролову и Олюнину – это очень трудоемкая работа.

Ответ: по работе мне приходится иметь дело с географическими картами в разных проекциях. Зачем же изображать одну и ту же местность в разных формах? Ответ очевиден – чтобы лучше отобразить имеющиеся данные. Есть справочник, а есть пособие. Первый предполагает, что его использует программист, осваивающий новый для себя язык. А второе – рассчитано и на тех, кто, может быть, только вообще начинает учиться по-настоящему программировать. В случае Фролова и Олюнина довольно полный справочник совмещен с процессом обучения, что и делает эту книгу ценным материалом, который хочется сохранить, очистив от давно отмерших вещей. Что же касается большой трудоемкости работы по написанию, то это было бы так, если бы я писал текст с нуля. А имея перед собой оцифрованную книгу, читать и менять текст легко. Получается что-то вроде программирования с шаблонами.

Даже легко находятся древние ошибки. Так, в примере классического решения квадратного уравнения в предпоследней строке страницы 102 в выражении DC=SQRT(COMPLEX(D,0)/(2*A)); последняя скобка должна была относиться только к корню, а не ко всей дроби. А на странице 190 приведен текстовый вывод структуры, одним из полей которой является указатель, что невозможно по определению.

Я искренне убежден, что если бы для персональных компьютеров в начале 80-х стартовой базой развития языков стал бы PL/1, а не Си (которому, кстати, уже полвека) или Паскаль (которому тоже за 50), то путь до современного уровня средств программирования был бы пройден быстрее и проще, поскольку не пришлось бы «открывать» многие вещи заново. Не пришлось бы сначала объявлять главным достоинством языков – их простоту и компактность, а затем десятилетиями добавлять в них изначально отсутствовавшие возможности и выдавать это за достижения, уже не упоминая про исходную простоту.

Поэтому главная цель этой попытки воссоздать старое пособие на современный лад – это познакомить программистов (особенно начинающих) с языком, о котором они возможно и не слышали, как с одной из альтернативных ветвей развития средств ИТ. И не только познакомить, но и предложить использовать.

Возможно, кто-то и начнет использовать это средство наряду с другими в своем арсенале, а для этого, по крайней мере, надо понять на простых примерах, что оно может. Вот тогда и потребуется данное пособие-справочник.

Итак, вперед по дороге, проложенной Геннадием Дмитриевичем и Виктором Юрьевичем.

ББК 22.18
Ф91
УДК 519.6 (075.8)

Фролов Г. Д., Олюнин В. Ю. **Практический курс программирования на языке PL/1.** — М.: Наука. Гл. ред. физ.-мат. лит., 1986.— 384 с.

Учебное пособие содержит описание алгоритмического языка PL/1, учитывающего особенности ЕС ЭВМ, с большим количеством примеров, на которых показывается, как должны пониматься и практически использоваться те или иные конструкции языка.

В конце каждого параграфа содержится набор контрольных вопросов, а также совокупность задач для закрепления изучаемого материала.

Описание языка дается весьма полно, и оно может быть использовано при программировании на языке PL/1.

Первое издание вышло в 1983 г.

Геннадий Дмитриевич Фролов, Виктор Юрьевич Олюнин

ПРАКТИЧЕСКИЙ КУРС ПРОГРАММИРОВАНИЯ

НА ЯЗЫКЕ PL/1

Редактор Л. Г. *Полякова*
Художественный редактор *Т. Н. Кольченко*
Технический редактор *И. Ш. Аксельрод*
Корректоры *Л. И. Назарова. Л. С. Сомова*

ИБ № 12824

Печать с матриц. Подписано к печати 10.10.86 г. Т-19621. Формат 60X90$^{1}/_{16}$. Бумага тип.
№ 2. Литературная гарнитура. Высокая печать. Усл.-печ. л. 24. Усл. кр.-отт. 24,25.
Уч.-изд. л. 29,92. Тираж 111 000 экз. Заказ № 3237. Цена 1 р. 30 к.

Ордена Трудового Красного Знамени издательство «Наука»
Главная редакция физико-математической литературы
117071 Москва В-71, Ленинский проспект, 15

Ордена Октябрьской Революции и ордена Трудового Красного Знамени МПО «Первая Образцовая типография» имени А. А. Жданова Союзполиграфпрома при Государственном комитете СССР по делам издательств, полиграфии и книжной торговли.

113054, Валовая, 28

1702070000-006
Ф---------- КБ-40-3-85
53(02)-87

Подготовка кадров в области прикладной математики и, в частности, в области программирования на современных вычислительных машинах — одна из центральных задач нашего общества.

Широкое распространение в различных отраслях народного хозяйства у нас в стране получили вычислительные машины единой системы, относящиеся к ЭВМ третьего поколения. Эти машины имеют весьма развитое программное обеспечение, основу которого составляют операционные системы (ОС ЕС ЭВМ).

Операционные системы позволяют увеличить пропускную способность вычислительной машины за счет обеспечения постоянной занятости всех ее компонентов; уменьшить время реакции вычислительной машины за счет уменьшения доли участия человека в управлении потоком выполнения прикладных программ; упростить разработку прикладных программ за счет использования большого количества эффективных языков программирования, большого набора обслуживающих программ, возможности накопления библиотеки прикладных программ и т. п. К числу обслуживающих программ относятся программы, обеспечивающие перевод прикладных программ с языка программирования на язык ЭВМ, отладку прикладных программ, обслуживание наборов данных и т. п.

На современном этапе составление прикладных программ выполняется, как правило, на языках программирования. Наибольшее распространение получили такие языки программирования, как алгол, фортран, кобол, PL/1. Язык PL/1 является наиболее развитым из числа применяемых на практике языков. Он включает в свой состав все то лучшее, что имеется в упомянутых выше языках.

Настоящая книга содержит достаточно полное описание языка PL/1 [1]), широкий состав примеров, вопросов и упражнений, позволяющих более глубоко понять и запомнить материалы соответствующих разделов. В книге рассмотрены две версии языка PL/1. Первая версия, PL/l (F), является входным языком стандартного компилятора PL/1 операционной системы ОС ЕС ЭВМ, называемого обычно компилятором уровня F. Вторая версия, PL/1 (O), является входным языком 3-й редакции оптимизирующего компилятора PL/1.

Книга может служить не только учебным пособием для изучающих язык PL/1, но и справочным руководством для программистов, работающих с этим языком.

Г. Д. Фролов, В. Ю. Олюнин

1) Не рассмотрены лишь некоторые специфические возможности ввода- вывода и несколько редко используемых встроенных функций.

Содержание

1. ПЕРВИЧНЫЕ КОНСТРУКЦИИ

1.1. Основные символы

Основные символы языка PL/1 делятся на следующие группы: буквы, цифры и специальные символы. В качестве *букв* используются прописные и строчные русские и латинские буквы и символы $ (знак денежной единицы), # (знак номера), @ (коммерческий знак *'at'),* _ (подчеркивание), ? (знак вопроса).

Цифра — это один из следующих десяти символов: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Специальный символ — это один из символов, перечисленных в табл. 1.1. В дальнейшем под буквенно-цифровым символом будем понимать букву или цифру.

Из символов языка образуются *слова* данного языка, которые являются минимальными конструкциями, имеющими в данном языке некоторый самостоятельный смысл. Эти слова могут обозначать числа, имена величин, некоторые действия и т.д.

В языке PL/1 различаются два типа слов: служебные (ключевые) слова и слова пользователя. Служебные слова имеют фиксированное начертание и раз и навсегда заданный смысл. Слова пользователя выбираются разработчиком программы по его усмотрению, но с соблюдением определенных правил, принятых в данном языке. Слова пользователя по своему начертанию могут совпадать со служебными словами. Служебные слова могут быть как английскими, так и русскими (каждое русское служебное слово имеет английский эквивалент и наоборот), и будут приводиться в процессе описания тех конструкций языка, в которых они могут встречаться.

В словах одинаковые по написанию русские и латинские буквы и строчные и прописные буквы считаются одной и той же буквой, если это не текстовая константа в апострофах.

П р о в е р ь с е б я

1. Какое количество основных символов используется в языке PL/1?

2. Входят ли в состав основных символов строчные латинские буквы *a, b, c, d, ...* русские буквы, синтаксические знаки русского языка: точка, запятая, точка с запятой, двоеточие?

3. На какие типы подразделяются слова в языке PL/1?

ТАБЛИЦА 1.1.

Специальные символы

Символ	Название символа	Символ	Название символа
*	**звездочка**	(	**открывающая скобка**
/	**косая черта**	)	**закрывающая скобка**
+	**плюс**	'	**апостроф**
-	**минус**	&	**амперсанд**
.	**точка**	=	**равно**
,	**запятая**	>	**больше**
;	**точка с запятой**	<	**меньше**
:	**двоеточие**	~	**отрицание**
|	**вертикальная черта *)**	^	**отрицание**
%	**процент**	␣	**пробел**

*) Вместо символа «вертикальная черта», допускается использование символа «!» (восклицательный знак) или «\» (обратная косая черта)

1.2. Типы величин

В программах на языке PL/1 могут использоваться величины следующих *типов:* целого, вещественного, комплексного (эти три типа называют *арифметическими* или *числовыми),* символьного, битового (эти два типа называют *строковыми).*

Значениями *целых* и *вещественных величин* являются целые и действительные числа. Они могут быть положительными, отрицательными и равными нулю. Значениями *комплексных величин* являются комплексные числа.

В качестве значений *величин символьного типа* могут выступать последовательности любых (даже не имеющих отображение на печати) символов. Значениями *величин битового типа* являются последовательности битов (обычно *бит* представляют цифрой 0 или 1).

Проверь себя

1. Какие типы величин используются в языке PL/1?
2. Какие числа могут быть использованы в качестве значений величин арифметического типа?
3. Что можно рассматривать в качестве значений величины символьного типа?

1.3. Константы

Константы используются в программах на языке PL/1 для обозначения постоянных значений величин; различают следующие типы констант: числовые (десятичные константы с фиксированной точкой, десятичные константы с плавающей точкой, мнимые константы), строковые (символьные и битовые — см. гл. 5).

Числовые константы могут быть действительными и мнимыми. К числу *действительных констант* относятся десятичные константы с фиксированной точкой, десятичные константы с плавающей точкой. Рассмотрим более подробно указанные типы констант.

Действительные числовые константы с фиксированной точкой могут иметь один из следующих видов:

а) $sa_1a_2 \ldots a_n \ (n \geq 1)$,

б) $sa_1a_2 \ldots a_n . b_1b_2 \ldots b_m \ (n \geq 0, m \geq 0, n+m \geq 1)$, где s —либо пусто, либо знак константы (плюс или минус), $a_i \ (i= 1, 2,\ldots, n)$ — десятичная цифра целой части, $b_j \ (j=1, 2, \ldots, m)$ — цифра дробной части. Символ . (точка) — разделитель целой и дробной частей константы. После знака константы допустимо наличие одного или нескольких пробелов.

Примеры десятичных констант с фиксированной точкой:

10 (обозначает число 10),
+15 (обозначает число 15),
-1.2 (обозначает число -1,2),
.01 (обозначает число 0,01).

Десятичную константу с фиксированной точкой, имеющую вид $sa_1a_2\ldots a_n$, будем в дальнейшем называть *целой десятичной константой.*

Десятичные константы с плавающей точкой могут иметь вид k_1Ek_2, где k_1 — любая десятичная константа с фиксированной точкой; k_2 — целая десятичная константа. Если десятичная константа соответствует числу x, а k_2 — числу y, то константа k_1Ek_2 соответствует числу $x \cdot 10^y$.

Примеры десятичных констант с плавающей точкой:

0E0 (обозначает число 0),
-1.2E+12 (обозначает число $-1{,}2 \cdot 10^{12}$),
.505E-2 (обозначает число 0,00505),
1E-50 (обозначает число 10^{-50}).

Мнимые константы имеют вид ai. Здесь a — любая действительная числовая константа. Мнимая константа обозначает комплексное число с нулевой действительной частью.

Примеры мнимых констант:

1i (обозначает число i),

0i (обозначает число 0),

1E8I (обозначает число 10^8i),

Комплексные числа *x+yi* и *x-yi,* где *x и y* — действительные числа, в рассматриваемой версии языка PL/1 представляются в виде текстовых строк (см. § 1.7) *'a+bi'* и *'a-bi'*, где *a* и *b* — действительные константы, обозначающие соответственно числа *x* и *y.*

П р о в е р ь с е б я

1. С какой целью в PL/1-программах используются числовые константы?
2. Сколько различают типов числовых констант, и какие?
3. Можно ли запись -.02760 рассматривать как числовую константу?
4. Какое число определяет числовая константа +.702E-2?
5. Какое комплексное число определяет константа -2.3E+2I?
6. Нижеприведенные числа представить в виде числовых констант.

1)-0.00000000704,

2) 6280000000.00,

3) +27,6 10^{-15},

4) 0,

5) $\sqrt{-1}$.

1.4. Идентификаторы, комментарии и пробелы

Идентификатор есть последовательность вида $ab_1 \ldots b_n$, где $0 \le n \le 30$, *a* — буква либо символ (вопросительный знак, подчеркивание, знак номера, знак «эт», знак денежной единицы); b_i $(i=1, 2, \ldots, n)$ — либо буква, либо цифра, либо символ (вопросительный знак, подчеркивание, знак номера, знак «эт», знак денежной единицы). Идентификаторы используются для обозначения различных объектов (переменных, массивов, функций и т.п.).

Примеры идентификаторов:

A счетчик sin #B#15 _time_23 очень_длинный_идентификатор

Последовательность символов 2a36 — не идентификатор, так как эта последовательность начинается с цифры. Идентификаторы относятся к словам пользователя и выбираются разработчиком программ по своему усмотрению. Хорошо выбранный идентификатор одновременно является и своеобразным комментарием к программе.

Комментарии и пробелы, как правило, используются программистом для наглядности текста программы и для упрощения в работе с программой. *Комментарий* есть последовательность вида /* $a_1a_2\ldots a_n$ */, где $0 \le n$; в подпоследовательность $a_1a_2\ldots a_n$ не входит ни одна из пар символов /* или */; a_i $(i=1, \ldots n)$ — любой символ.

Примеры комментариев:

а) /* ФУНКЦИЯ SIN (X)=Y */

б) /* TEXT */

в) /* COS(X)=Y; SIN (X)=Z ПРИ X=0.037 */

Однострочные комментарии // $a_1a_2\ldots a_n$, начинаются с символов // и действуют только до конца строки.

Примеры однострочных комментариев:

а) // повторный расчет

б) // /*TEXT */

в) // COS(X)=Y; SIN (X)=Z ПРИ X=0.037

Псевдографические символы как пробелы. Если псевдографический символ встречается в тексте не внутри символьной константы, он считается пробелом, поэтому можно с помощью таких символов структурировать текст, например:

| x=0; | эквивалентно записи x=0;

Комментарии и пробелы могут быть использованы между любыми конструкциями PL/1-программы, но не внутри идентификаторов, констант и составных символов (*составными символами* в языке PL/1 называют следующие пары символов: **, ||, */, /*, //, ->, ^=, ^>, ^<, >=, <= , <=>).

П р о в е р ь с е б я

1. С какой целью в PL/1-программе используются идентификаторы?
2. Может ли идентификатор содержать 32 символа и начинаться с цифры?
3. Может ли идентификатор содержать буквы русского алфавита?
4. Какие из приведенных ниже записей можно рассматривать как идентификаторы:

1)	A02IB	4) a+b=C	7) $IFFA464
2)	3fAB	5) Ira	8) номер-1
3)	VA'CD	6) sum	9) x1-x2

5. Какое количество различных идентификаторов, которые можно образовать из а) символа *X* и б) символов *A* и *1*.

6. Указать, какие из приведенных ниже строк символов можно рассматривать как комментарий:

1) / * МАМА */
2) I* *1
3) /* перейти к функции Y=F(X, Z) */
4) // ОСТАНОВ */
5) /* ошибка при X<0 /* иначе передать управление
/* на */

7. В каких конструкциях языка PL/1 не допускается использование комментариев и пробелов?

1.5. Простая переменная

Переменная — это величина, которая может принимать различные значения. Переменные могут быть объединены в совокупности — массивы и структуры (см. гл. 3 и 7). Переменную, не входящую ни в какую совокупность, называют *простой.* Простая переменная в PL/1-программе представляется идентификатором.

Переменные в языке PL/1 делятся на следующие *типы: арифметический* (вещественный или комплексный), *строковый* (символьный или битовый), *управляющий.* Переменная арифметического типа принимает числовые значения, строкового типа — строковые значения. Переменные управляющего типа могут быть адресными или файловыми. Конкретная переменная может принимать значения только одного типа.

Каждой переменной явно или по умолчанию приписываются определенные свойства. К свойствам арифметической (числовой) переменной относятся: тип (вещественный или комплексный), форма представления (с фиксированной или с плавающей точкой), основание (десятичное или двоичное) и точность значений переменной. Если значения переменной вещественного типа представляются в форме с плавающей точкой, то их точность задается одним параметром *(p),* который указывает, что в памяти должны сохраняться, по крайней мере, p значащих (двоичных или десятичных) цифр каждого значения переменной. Если же значения переменной вещественного типа представляются в форме с фиксированной точкой, то их точность задается двумя параметрами *(p, q).* Эти параметры указывают, что без искажения могут быть сохранены такие значения переменной, которые равны $a \cdot t^q$, где a — целое число, $|a| < t^p$, t равно 2 или 10

в зависимости от системы счисления, в которой представляются значения переменной.

Для переменной комплексного типа задается единая точность представления коэффициентов при действительной и мнимой частях комплексного числа, которое является значением переменной. Эта точность задается по тем же правилам, что и точность значений переменных вещественного типа. Комплексные переменные могут иметь только форму с плавающей точкой.

П р о в е р ь с е б я

1. Что принято рассматривать в языке PL/1 в качестве переменной?
2. Чем представляется простая переменная в PL/1-программе?
3. На какие типы делятся переменные?
4. Перечислить свойства арифметических (числовых) переменных.
5. Каким образом задается точность значений числовых переменных?

1.6. Указатели функций

В языке PL/1 с понятием функции связано два понятия — это указатель функции и подпрограмма (алгоритм) вычисления функции. *Указатель функции* — средство обращения к алгоритму вычисления функции. Указатель функции обычно имеет вид

$$f\ (x_1,\ x_2,\ \ldots,\ x_n)$$

где f — *имя функции*; $x_i\ (i=1,\ 2,\ \ldots,\ n)$ — ее аргумент, называемый также *фактическим параметром* функции. В качестве имен функции используются идентификаторы, а в качестве фактических параметров — выражения (в частности, константы, переменные).

Свойства функции задаются программистом явно или по умолчанию (к свойствам числовых функций относятся тип, форма представления, основание и точность значения функции). Указатель функции автоматически наделяется всеми теми свойствами, которыми обладает и сама функция.

Примеры указателей функций:

```
q(3.7)          интеграл(-0.05, +6.85)
sin(X)          f2a(x,z/3, y)
```

Указатель функции используется в выражениях. Наличие указателя функции в выражениях указывает на обращение к подпрограмме вычисления соответствующей функции. После того как все необходимые вычисления по

этой подпрограмме будут выполнены, значение функции присваивается данному указателю функции.

П р о в е р ь с е б я

1. Какие понятия языка PL/1 связаны с понятием функции?
2. Что используется в указателе функции в качестве имени функции и аргумента?
3. Можно ли рассматривать приведенные ниже записи в качестве указателей функций, если *x* — выражение?

1) sin)
2) cos(x)
3) X(x)
4) 2X(x)
5) fixed(x)
6) sin(x

1.7. Выражение

Выражение образуется из операндов, знаков операций и круглых скобок. В качестве знаков операций используются символы или совокупность символов: + (сложить), - (вычесть), * (умножить), / (разделить), ** (возвести в степень), = (равно), ^= (не равно), > (больше), ^> (не больше), >= (больше либо равно), < (меньше), ^< (не меньше), <= (меньше либо равно), ^ или ~ (логическое отрицание), & (логическое умножение), | (логическое сложение), || (сцепление строк).

Все операции разбиваются на одноместные и двуместные. К *одноместным операциям* относятся: + (одноместная операция «плюс»), - (одноместная операция «минус»), ^ (логическое отрицание). К *двуместным операциям* относятся все остальные.

Операндами выражения могут быть константы, переменные, указатели функций, в некоторых случаях массивы или структуры. В настоящей главе мы рассмотрим выражения в качестве операндов, у которых могут быть константы, переменные, указатели функций. Действия над массивами и структурами рассмотрим в главах 3, 7.

Простейшее выражение состоит из одного операнда. Примерами таких выражений являются:

A *3.1415926* *f(x)*

Здесь *f(x)* — указатель функции, в котором *f* — имя функции, *x* — фактический параметр (аргумент функции), *A* — имя переменной.

Более сложные выражения состоят из нескольких операндов, соединенных знаками двуместных операций. Примерами таких выражений являются:

x + y
*3.5+7*k - z*
*sqrt(x)**2 + c15*f(3)*

Здесь *sqrt(x)* и *f(3)* — указатели функций; *x, y, z, k* и *c15* — имена переменных. Перед операндом могут стоять знаки одноместных операций, как, например, в следующих выражениях:

-sin(x)
-6+a**k-C15
^z & a > 0.7

Любое выражение может быть заключено в круглые скобки. Выражения в скобках могут быть использованы на правах операндов для составления более сложных выражений. Например, если *a*, *b* и *c* — имена переменных, то в качестве выражений можно рассматривать:

(a +b+c)
-(a-2)
(a - b)*(b - c)
a + 3*(a - (b + c)**2)
2**- (a + b)

Порядок, в котором выполняются операции, определяется наличием в выражении скобок и старшинством *(приоритетом)* операций. Установлен следующий порядок старшинства операций:

1) одноместные операции (знаки операций: +, - и ^), и возведение в степень (знак операции **);
2) умножение (знак операции *) и деление (знак операции /);
3) сложение (знак операции +) и вычитание (знак операции -);
4) сцепление строк (знак операции ||);
5) сравнения (знаки операций: =, >, ^>, ^<, ^=, <= и >=);
6) логическое умножение (знак операции &);
7) логическое сложение (знак операции |).

Операции ** и ^ и одноместные операции + и -, следующие в выражении непосредственно друг за другом, выполняются в последовательности справа налево. Если операции одного и того же старшинства (за исключением указанных) следуют в выражении друг за другом, то они выполняются в последовательности слева направо.

Например, в выражении x**y**2 последовательность выполнения операций будет следующая: 1) y**2 (обозначим результат операции через z),

2) x**z; тогда как в выражении x+y+0.765 операции будут выполняться в следующем порядке: 1) x+y (обозначим результат операции через z), 2) z+0.765.

Отметим, что не всегда математическое выражение без скобок может быть представлено бесскобочным выражением языка PL/1. Например, выражение $\frac{a}{bc}$ должно быть записано в виде A/(B*C), а не в виде A/B*C, соответствующем выражению $\frac{a}{b}$ с

Все операции в языке PL/1 разбиты на четыре класса: арифметические операции, операции сравнения, логические операции и операция сцепления.

Арифметические операции. К *арифметическим операциям* относятся: сложить (+), вычесть (-), умножить (*), разделить (/), возвести в степень (**). Эти операции непосредственно определены только для арифметических операндов, к числу которых относятся: числовая константа, переменная и указатель функции, принимающие числовые значения. Арифметические операнды двуместной операции могут иметь различные свойства (характеристики): тип (вещественный или комплексный), основание (десятичное или двоичное), форма представления (с плавающей или фиксированной точкой), точность *(p)* для чисел с плавающей точкой и *(p, q)* для чисел с фиксированной точкой.

Если характеристики операндов не совпадают, то перед выполнением арифметической операции один или оба операнда преобразуются в соответствии с правилами:

1. Если один из операндов является двоичным числом, а другой — десятичным, то последний преобразуется к двоичному основанию.
2. Если один из операндов является числом с плавающей точкой, а другой — числом с фиксированной точкой, то последний преобразуется к форме с плавающей точкой.
3. Если один из операндов является комплексным числом, а другой — вещественным числом, то последний преобразуется в комплексное число.

Точность результата преобразования формы представления и (или) основания определяется по следующим формулам:

1) число с фиксированной точкой и точностью (p_1, q_1) в число с плавающей точкой с тем же основанием и точностью (p_2): $p_2=p_1$

2) десятичное число с точностью (p_1) или (p_1, q_1) в двоичное число с плавающей точкой и точностью (p_2): $p_2 = min\ (N, [3, 32\ p_1]\ )$

где N равно 53 или 24; $[x]$ здесь и ниже обозначает ближайшее к x целое число, не меньшее чем x по абсолютной величине;

Для числовых констант параметр p точности равен общему количеству цифр в константе (цифры порядка в константах с плавающей точкой не учитываются), а параметр q — количеству цифр в дробной части константы.

При преобразовании десятичного числа с фиксированной точкой (как в двоичное число с любой формой представления, так и в десятичное число с плавающей точкой) не исключено изменение этого числа. При этом следует учитывать такую особенность числовых преобразований в языке PL/1, как отсутствие каких-либо правил округления — лишние младшие цифры обычно отбрасываются независимо от их значений.

Результатом преобразования действительного числа в комплексное является комплексное число с нулевой мнимой частью и коэффициентом при действительной части, равным исходному числу. В результате выполнения арифметических операций получаются числа с тем же основанием и формой представления, что и у окончательных значений операндов операций; если операнды — комплексные числа, то и результат операции будет комплексным числом. Точность результатов операций определяется по следующим правилам:

1. Результат одноместной операции имеет ту же точность, что и операнд.
2. Если операнды любой двуместной операции — числа с плавающей точкой и точностью p_1 и p_2, то точность результата будет равна $max\ (p_1, p_2)$.
3. При сложении или вычитании чисел с фиксированной точкой и точностью (p_1, q_1) и (p_2, q_2) точность результата будет равна (p, q), где

$$p=min(1+max(p_1 - q_1, p_2 - q_2)+max(q_1, q_2), N)$$

$$q= max\ (q_1, q_2).$$

Здесь и ниже N равно 63 или 31 для двоичных чисел и 15 для десятичных.

4. При умножении чисел с фиксированной точкой и точностью (p_1, q_1) и (p_2, q_2) точность результата будет равна (p, q), где

$$p=min(l+p_1+p_2, N) \qquad q=q_1+q_2.$$

5. При делении чисел с фиксированной точкой и точностью (p_1, q_1) и (p_2, q_2) точность результата будет равна (p, q), где

$$p= N \qquad q= N-(p_1 - q_1+q_2)$$

6. Если второй операнд операции возведения в степень (показатель степени) представлен целой константой без знака со значением, равным n, а первый операнд имеет значение в форме с фиксированной точкой и точностью

(p_1, q_1), причем $n*(p_1+1)-1 \leq N$, где N равно 15 при десятичном основании значения первого операнда и 63 или 31 — при двоичном основании, то результат операции будет числом в форме с фиксированной точкой с тем же основанием, что и первый операнд, и с точностью (p, q), где

$$p=n*(p_1+1)-1 \qquad q=n*q_1$$

7. Если второй операнд операции возведения в степень имеет вещественное значение в форме с фиксированной точкой и точностью (p, q), где $q=0$, и при этом шестое правило неприемлемо, то значение первого операнда при необходимости преобразуется к форме с плавающей точкой, а результат операции будет всегда числом с плавающей точкой с тем же основанием и той же точностью, что и у первого операнда.
8. Если шестое и седьмое правила неприемлемы, то значения обоих операндов при необходимости преобразуются к форме с плавающей точкой и к одному основанию; точность результата определяется так же, как и для других операций.

Смысл арифметических операций в языке PL/1 соответствует общепринятому смыслу. Однако на выполнении операции возведения в степень в рассматриваемой версии языка PL/1 имеется следующее ограничение. Если окончательное значение второго операнда операции возведения в степень представлено в форме с плавающей точкой, и оба операнда являются действительными числами, то первый операнд не может быть отрицательным числом независимо от значения показателя степени. Например, если x — переменная с вещественными значениями в форме с плавающей точкой и ее текущее значение равно -2, то выполнение операции $x**x$ приведет к аварийному завершению программы (несмотря на то, что -2^{-2} равно 0,25).

Результат выполнения арифметической операции может оказаться за пределами, допустимыми для чисел данного типа. Например, если N — переменная с двоичными значениями в форме с фиксированной точкой и точностью (31, 0) и ее текущее значение равно 10_000_000, то результат операции $N*N$, равный 10^{14}, не представим в виде; двоичного числа в форме с фиксированной точкой с точностью (31, 0). В подобных случаях выполнение программы аварийно завершается (если не использованы специальные средства языка, рассмотренные в гл. 8).

Операнды арифметических операций могут относиться к строковому типу. В этом случае перед выполнением операции строковый операнд преобразуется в арифметический по правилам, рассмотренным в гл. 5.

Операции сравнения. К *операциям сравнения* относятся: = (равно), ^= (не равно), > (больше), ^> (не больше), >= (больше либо равно), < (меньше), ^< (не меньше), <= (меньше либо равно). Все эти операции являются двуместными. Результатом операции является строка из одного бита, равного 1, если сравнение верно для данных операндов, и равного 0 — в противном случае.

Все операции сравнения определены для арифметических операндов вещественного типа. Для арифметических операндов комплексного типа определены лишь две операции сравнения: = (равно) и ^= (не равно). Операции сравнения для арифметических операндов имеют общепринятый смысл.

Если операнды относятся к арифметическому типу и имеют разные характеристики, то перед выполнением операции сравнения производится преобразование операндов по тем же правилам, что и для арифметических операций. Правила выполнения операций сравнения в случае, когда один или оба операнда относятся к строковому типу, рассмотрены в гл. 5.

Логические операции. К *логическим операциям* (*И, ИЛИ, НЕ*) относятся & *(логическое умножение)*, | *(логическое сложение)*, ^ *(логическое отрицание)*. Логические операции определены для операндов, имеющих значение битового типа. Результаты выполнения логических операций над строками, состоящими из одного бита, приведены в табл. 1.2.

ТАБЛИЦА 1.2.

Правила выполнения логических операций

A	1	1	0	0
B	1	0	1	0
^A	0	0	1	1
A&B	1	0	0	0
A | B	1	1	1	0

Правила выполнения логических операций над строками, состоящими более чем из одного бита, рассмотрены в гл. 5. Там же рассмотрены правила преобразования операндов логических операций, не относящихся к битовому типу. Вместо символов &, |, ^ можно использовать их русские эквиваленты: И, ИЛИ, НЕ.

Операция сцепления. *Операция сцепления* (знак ||) определена для операндов строкового типа. Правила выполнения этой операции рассмотрены в гл. 5.

П р о в е р ь с е б я

1. Перечислите знаки операций, применяемые в выражениях.

2. Пусть A и B — имена переменных. Какие из нижеприведенных записей можно рассматривать как выражения языка PL/1?

1) -3,14	6) A+B	11) (A)
2) +2.81	7) A* - B	12) (- (A - B))
3) -A	8) ((A + B)	13) A + ^B|C
4) A-	9) A//B	14) A= B =0
5) 2A	10) A || B	15) A= ^= B

3. Представьте в виде выражений следующие алгебраические выражения:

$$\frac{x^2+y^2}{2z} \qquad \frac{\sqrt{x}+\sqrt{y}}{5i} \qquad \frac{\sqrt[3]{x}+\sqrt[3]{y}}{2-3z}$$

Пусть имеются следующие переменные:

A — комплексного типа, значение которой представлено в форме с плавающей точкой и точностью (24);

B — вещественного типа, значение которой представлено в двоичной системе счисления в форме с плавающей точкой и точностью (53);

C — вещественного типа, значение которой представлено в десятичной системе счисления в форме с фиксированной точкой и точностью (10, 1);

D — вещественного типа, значение которой представлено в десятичной системе счисления в форме с плавающей точкой и точностью (10).

Укажите:

а) к какому виду будет преобразовано значение каждого операнда следующих операций:

1) A + B 2) B+D 3) A + C 4) C + D 5) A + D

б) характеристики результатов операций:

1) A + C	4) B*D	7) A**2
2) B—D	5) A/C	8) B**2
3) A*C	6) B/D	9) B**D

4. Пусть переменная *x* имеет вещественные значения в форме с плавающей точкой. При каких из ниже приведенных значений *x* операция *x****x* недопустима:

1) 2 2) 0 3) 0,2 4) -0,2 5) -1

5. Указать порядок выполнения операций в следующих выражениях:

1) x2a**y**(3 + x)
2) 0.34 + 2*(x + y) - x/(x + y)*2.74
3) (a - 17.3*x)^<0.5& ^y|b
4) a&b|x^>(y- x)*2**y-0.37

6. Определить основание, форму представления и точность (количество цифр и, если необходимо, масштабный множитель) результата вычисления следующих выражений:

1) L*(M+K*(I + K))-2 если I, K, M и L — переменные с десятичными значениями с фиксированной точкой и точностью (5,0), (1,0), (2,1) и (1,1) соответственно;

2) (a**3)**2+a/2 если a — переменная, принимающая десятичные значения с фиксированной точкой и точностью (5, 5);

3) (a-1)*(b-0.1) если a и b — переменные, принимающие десятичные значения с фиксированной точкой и точностью (5, 1) и (5, 2) соответственно.

1.8. Встроенные функции для обработки числовых данных

В языке PL/1 имеется большое количество так называемых *встроенных функций*. Свойства этих функций, а также алгоритмы вычисления их значений, всегда считаются заданными по умолчанию. Рассмотрение встроенных функций будем проводить по мере изучения тех возможностей языка, с которыми связано использование этих функций.

Рассмотрим встроенные функции, предназначенные для обработки числовых данных. Эти функции можно разделить на 4 группы: элементарные числовые функции, трансцендентные числовые функции, функции преобразования чисел и функции, управляющие точностью вычисления арифметических операций. При описании точности значений функций через *N* обозначено число 15 для десятичных значений и 31 или 63 — для двоичных значений функций; через p и q — характеристики точности значения функции; через p_1 q_1 p_2 q_2, … p_n q_n — характеристики точности окончательных значений аргументов функции.

К встроенным элементарным числовым функциям относятся функции: ABS(X), CEIL(X), COMPLEX(X), CONJG(X), FLOOR(X), IMAG(X), MAX(X,Y), MIN(X, Y), MOD(X, Y), REAL(X), ROUND(X, Y), SIGN(X), TRUNC(X).

Встроенная функция **ABS(x)** предназначена для вычисления значения функции

$$\text{ABS(x)}=\begin{cases}|x| & \text{если x имеет вещественное значение,}\\ +\sqrt{a^2+b^2} & \text{если x имеет комплексное значение } a+bi.\end{cases}$$

Здесь x — выражение. Значение функции ABS(x) является вещественным числом с той же формой представления, основанием и точностью, что и у аргумента.

Встроенная функция **CEIL(x)** предназначена для вычисления значения функции CEIL(x)=n

где x — выражение, принимающее вещественные числовые значения, n — наименьшее целое число, большее либо равное значению x $(0 \le n\text{-}x \le 1)$. Значение функции — вещественное число с той же формой представления и основанием, что и у аргумента. Точность значений функции в форме с плавающей точкой совпадает с точностью аргумента, а точность значения функции в форме с фиксированной точкой равна:

$$p=min(max(p_1 - q_1+1,1), N) \qquad q= 0$$

Встроенная функция **COMPLEX(x, y)** предназначена для вычисления значения функции COMPLEX(x, y)$=a+bi$

где x и y — выражения, принимающие соответственно значения вещественных чисел a и b. Форма представления, основание и точность значения функции определяются теми же правилами, что и для суммы чисел $a+b$ (см. § 1.7).

Встроенная функция **CONJG(x)** предназначена для вычисления сопряженного значения функции CONJG(x)=a-bi

где x — выражение, принимающее значение комплексного числа $a+bi$. Форма представления, основание и точность значения функции такие же, как и у аргумента.

Встроенная функция **FLOOR(x)** предназначена для вычисления значения функции FLOOR(x)=n

где x — выражение, принимающее вещественные значения, n — наибольшее целое число, не превышающее значения x $(0 \le x\text{-}n < 1)$. Характеристики значения функции FLOOR (x) определяются тем же набором правил, что и для функции CEIL(x).

Встроенная функция **IMAG(x)** предназначена для вычисления значения функции IMAG (x) =b

где x — выражение, принимающее значение комплексного числа $a+bi$, a и b — вещественные числа. Форма представления, основание и точность значения функции такие же, как и у аргумента.

Встроенная функция **MAX(x1, x2)** предназначена для вычисления значения функции MAX(x_1,x_2) = *max a*

где x_1 и x_2 — два выражения, принимающее значение вещественного числа a.

Перед вычислением значения функции значения обоих аргументов преобразуются к одной форме представления и одному основанию. Форма представления и основание значения функции MAX — такие же, как у окончательных значений аргументов, а его точность равна $p=max(p_1, p_2)$ для чисел в форме с плавающей точкой и

$$p = min(max(p_1 - q_1, p_2 - q_2) + max(q_1, q_2), N)$$

$$q = max(q_l\ q_2)$$

для чисел с фиксированной точкой.

Встроенная функция **MIN(x1, x2)** предназначена для вычисления значения функции MIN $(x_1\ x_2) = min\ a$

где x_1 и x_2 — два выражения, принимающее значение вещественного числа a. Перед вычислением значения функции значения обоих аргументов приводятся к одному основанию и одной форме представления. Характеристики значения функции определяются так же, как и для функции MAX(x_1,x_2).

Встроенная функция **MOD(x, y)** предназначена для вычисления значения функции MOD $(x, y)=mod_y\ x=z$ т.е. остаток от деления по модулю

где x и y — выражения, принимающие вещественные числовые значения, z — вещественное неотрицательное число, равное остатку от деления нацело x на y ($0 \le z < |y|$ и $(x-z)/y$ — целое число). Перед вычислением функции значения аргументов приводятся к одному основанию и одной форме представления. Форма представления и основание значения функции MOD (x, y) такие же, как у окончательных значений аргументов, а его точность равна $p=max(p_1, p_2)$ для чисел с плавающей точкой и

$$p=min(p_2—q_2+max(q_1\ q_2), N) \qquad q=max(q_1\ q_2)$$

для чисел с фиксированной точкой.

Встроенная функция **REAL(x)** предназначена для вычисления значения функции REAL(x)=a

где x — выражение, принимающее значение комплексного числа $a+bi$, a и b — вещественные числа. Форма представления, основание и точность значения функции — такие же, как и у аргумента.

Встроенная функция **ROUND(x, y)** предназначена для вычисления значения функции ROUND $(x, y)=c$. имеется русский эквивалент названия **ОКРУГЛИТЬ(x, y)**

где x — выражение, принимающее вещественные или комплексные числовые значения, y — целое десятичное число, c — число, представляющее результат округления значения выражения x. Причем для чисел в форме с фиксированной точкой аргумент y задает позицию цифры, до которой округляется исходное число. Если $y \geq 0$, то это будет y-я цифра дробной части числа (считая слева направо); если же $y \leq 0$, то это будет $(-y + 1)$-я цифра целой части числа (считая справа налево). Например, значение ROUND/ОКРУГЛИТЬ (13.55, 0) равно 14. Точность значения функции в форме с фиксированной точкой равна $p=max(1, min(p_1-q_1+y+1, N))$ $q=y$

Для аргументов в форме с плавающей точкой округление состоит в присваивании единицы младшему биту внутреннего представления числа (независимо от y). Точность значения функции при этом равна точности аргумента. Особенностью применения функции ROUND/ОКРУГЛИТЬ к числам с плавающей точкой является то, что в среднем в половине случаев абсолютная величина значения функции будет больше абсолютной величины значения аргумента на минимально возможное при данной точности число, в остальных случаях значение функции будет равно значению аргумента. Так как при переходе от чисел с плавающей точкой с большей точностью к числам с меньшей точностью лишние цифры отбрасываются без округления, то, следовательно, многократное применение функции ROUND (x, y) в подобных случаях позволяет в среднем уменьшить регулярную ошибку вычислений.
Форма представления и основание счисления результата вычисления функции совпадает с формой представления и основанием счисления аргумента x. Смысл использования функции для битовых строк приведен далее.

Встроенная функция **SIGN(x)** предназначена для вычисления значения функции SIGN(x)=sign(x)= $\begin{cases} 1 \text{ при } x > 0 \\ 0 \text{ при } x = 0 \\ -1 \text{ при } x < 0 \end{cases}$

где x — выражение с вещественным числовым значением. Значение функции всегда является вещественным двоичным числом с фиксированной точкой и точностью (1, 0).

Встроенная функция **TRUNC(x)** предназначена для вычисления значения функции TRUNC(x)=n

где x — выражение с вещественным числовым значением, n — ближайшее к x целое число, не меньшее, чем x по абсолютной величине $(0 \leq |n-x| = |n|-|x| < 1)$. Характеристики значения функции TRUNC(x) определяются так же, как и для функции CEIL(x).

Трансцендентные числовые встроенные функции языка PL/1 соответствуют ряду общепринятых в математике трансцендентных функций действительной и комплексной переменных. Значение таких встроенных функций вычисляется с точностью 53 или 24 двоичных разряда мантиссы. Для многозначных функций комплексной переменной вычисляется их главное значение. Аргументами встроенных функций данной группы могут быть выражения с числовыми значениями с любой формой представления, любым основанием и любой точностью. Однако значения аргументов в форме с фиксированной точкой всегда перед вычислением функции преобразуются к форме с плавающей точкой (значения аргументов функций ATAN2 (x, y) и ATAND2 (x, y), кроме того, приводятся к одному основанию).

Указатели функций могут быть написаны и малыми буквами, записи SQRT(X) и sqrt(x) эквивалентны, записи FIXED(X) и fixed(x) эквивалентны. У английских ключевых слов BINARY, DECIMAL, FIXED, FLOAT имеются русские эквиваленты: **ДВОИЧНОЕ**, **ДЕСЯТИЧНОЕ**, **ТОЧНОЕ** и **ВЕЩЕСТВЕННОЕ** (или **ВЕЩ**) соответственно.

Значение любой из трансцендентных функций является комплексным или вещественным числом (в соответствии с аргументом функции) в форме с плавающей точкой, с тем же основанием и точностью, как и у окончательных значений аргументов (точность значения функций ATAN2(x, y) и ATAND2(x, y) равна $max(p_1\ p_2)$). Список всех встроенных функций данной группы приведен в табл. 1.3. Отметим, что значения функций SQRT(x) и EXP(x) вычисляются быстрее и точнее, чем, соответственно, значения выражений x**y и x**z, где *y* равно 1/2, а *z* равно числу *e* с любой возможной точностью.

ТАБЛИЦА 1.3.

Трансцендентные встроенные функции

Указатель функции	Значение функции	Допустимы ли комплексные аргументы
ACOS(x)	arccos *x*	да
ACOSD(x)	$\frac{180}{\pi}$arccos x, ответ в градусах	нет
ASIN(x)	arcsin x	да
ASIND(x)	$\frac{180}{\pi}$arccos x, ответ в градусах	нет
ATAN(x)	arctg x	да
ATAN2(x, *y)*	arctg (x/*y)* при *y* > 0 π + arctg (x/v) при x ≥ 0, v < 0 -π +-arctg (*x/y*) при x < 0. v < 0 π/2 при *x* > 0, v=0 -π/2 при x < 0, *y*= 0	нет
ATAND (x)	$\frac{180}{\pi}$arctg x, ответ в градусах	нет
ATAND2(x, *y)*	$\frac{180}{\pi}$arctg(x/y), ответ в градусах	нет
COS(x)	cos *x*	да
COSD(x)	cos($\frac{\pi x}{180}$) значение *x* представлено в градусах	нет
COSH(x)	ch x	да
COSSIN(x)	Возвращает cos x+ sin x i в комплексную переменную	нет
COSDSIND(x)	Возвращает cos($\frac{\pi x}{180}$)+ sin($\frac{\pi x}{180}$) i в комплексную переменную	нет
ERF(x)	$\frac{2}{\sqrt{\pi}}\int_0^x e - t2\ dt$	нет
ERFC(x)	$1 - \frac{2}{\sqrt{\pi}}\int_0^x e - t2\ dt$	нет
EXP(x)	e^x	да
LOG(x)	ln x	да
LOG2(x)	$\log_2$ x	нет
LOG10(x)	lg x	нет
SIN(x)	sin *x*	да
SIND(x)	sin($\frac{\pi x}{180}$). значение *x* представлено в градусах	нет
SINH(x)	sh x	да
SQRT(x)	$\sqrt{x}$	да
TAN(x)	tg x	да
TAND(x)	tg($\frac{\pi x}{180}$), значение *x* представлено в градусах	нет
TANH(x)	th x	да

ТАБЛИЦА 1.4.

Встроенные функции преобразования чисел

Указатель Функции	Характеристики значения *x*	Характеристики значения функции
BINARY(x, p)	в форме с фиксированной точкой	в форме с фиксированной точкой, с двоичным основанием и точностью *(p,* 0)
BINARY(x, p)	в форме с плавающей точкой	в форме с плавающей точкой, с двоичным основанием и точно*стью* (p)
BINARY(x)	с десятичным основанием	в той же форме, что и аргумент, с двоичным основанием и точностью [3,32p] или [1+3,32p], [3,32q]
DECIMAL(x, p, q)	в форме с фиксированной точкой	в форме с фиксированной точкой, с десятичным основанием и точ*ностью* (p, q)
DECIMAL(x, p)	в форме с фиксированной точкой	в форме с фиксированной точкой, с десятичным основанием и точностью (p, 0)
DECIMAL(x, p)	в форме с плавающей точкой	в форме с плавающей точкой, с десятичным основанием и точностью (p)
DECIMAL (x)	с двоичным основанием	в той же форме, что и аргумент с десятичным основанием и точностью [p/3,32] или [l+p/3,32], [q/3,32]
FIXED(x, p, q)	с двоичным основанием	в форме с фиксированной точкой, с тем же основанием, что и аргумент, и с точностью (p, q)
FIXED (x, p)	с двоичным основанием	то же, но с точностью (p, 0)
FIXED(x)	с двоичным основанием	то же, но с точностью (15, 0) для двоичных чисел и (5, 0) для десятичных чисел
FLOAT(x, p)	с двоичным основанием	в форме с плавающей точкой, с тем же основанием, что и аргумент, и точностью (p)
FLOAT(x)	с двоичным основанием	то же, но с точностью (6) для десятичных чисел и (24) для двоичных чисел

Встроенные функции, позволяющие преобразовывать форму представления, основание и точность чисел, перечислены в табл. 1.4. Аргументом *x* каждой из этих функций может быть выражение, принимающее значение вещественного числа. Значением функции может также быть либо

комплексное, либо вещественное число. Причем значением функции будет комплексное число, если аргумент этой функции — комплексное число, и вещественное число, если аргумент — вещественное число. Аргументы p и q должен быть целыми десятичными беззнаковыми константами, причем значения аргументов p и q не должны выходить за пределы, допустимые для параметров точности переменных соответствующего типа.

Для имен функций BINARY, DECIMAL и ВЕЩЕСТВЕННОЕ допустимы соответственно сокращения **BIN**, **DEC** и **ВЕЩ**. В описаниях указателей функций BINARY(x) и DECIMAL(x) выражение в квадратных скобках *[a]* обозначает ближайшее к a целое число, не меньшее, чем a, по абсолютной величине (при условии, что оно не выходит за пределы, допустимые для соответствующего параметра точности).

К встроенным функциям, управляющим точностью арифметических операций, относятся еще две функции:

MULTIPLY(x, y, p, q)= x•y или MULTIPLY(x, y, p) = x•y,
DIVIDE(x, y, p, q) = x/y или DIVIDE(x, y, p) = x/y.

Здесь x и y — выражения, принимающие значения вещественного типа; требования к аргументам p и q — те же, что и для функций числовых преобразований. Перед вычислением функции значения аргументов x или y могут быть автоматически, преобразованы по тем же правилам, что и для операндов арифметических операций. Характеристики значения функции (кроме точности) совпадают с окончательными характеристиками значений аргументов. Точность значения функции с фиксированной точкой равна (p, q) или $(p, 0)$, если q опущено. Точность значения с плавающей точкой равна (p), аргумент q при этом должен отсутствовать. У слов MULTIPLY и DIVIDE есть русские эквиваленты **УМНОЖИТЬ** и **ДЕЛИТЬ.**

Проверь себя

1. Какие функции называются встроенными?
2. В чем состоит эффект применения встроенной функции ROUND к числам с плавающей точкой?
3. Составьте выражения на языке PL/1 для следующих математических выражений:
   1) $\cos\frac{x}{y} \cdot \sin y^2 - tg^2 z$ (x, y и z — в радианах);
   2) $\cos xy \cdot \sin^2 y - tg\, z^2$ (x, y и z — в градусах);
   3) $ch\frac{x}{y} \cdot sh^2 y + th\, z^2$;
4. binary(float(2.2, 2) какой результат получится в итоге?

2. ОСНОВНЫЕ ОПЕРАТОРЫ, ПРОГРАММА С ПРОСТЕЙШЕЙ СТРУКТУРОЙ

2.1. Общий вид оператора и простейшей программы

Основной конструкцией PL/1-программы является *оператор*. В общем виде оператор можно представить следующим образом:

ms;

Здесь m — либо пусто, либо конструкция вида $m_1: m_2: .. .: m_k$ $(k \geq 1)$, где m_i $(i = 1, 2, \ldots, k)$ — идентификатор, называемый *меткой* оператора; s — собственно оператор (иногда s называют *телом* оператора); и конечная точка с запятой — признак конца оператора. Метки операторов, как правило, не должны совпадать с именами переменных (подробнее это ограничение рассмотрено в § 4.4).

Операторы используются для указания некоторых действий, описания свойств величин, определения структуры программы. Единственным оператором, не задающим никаких сведений, является *пустой оператор*. Тело этого оператора не содержит никаких символов. Например, пустой оператор с метками Ml и M2 имеет вид Ml: M2: ;

Программа с простейшей структурой представляет собой последовательность

$S_1S_2 \ldots S_{k-1}S_k$ где S_i $(i = 1, 2, \ldots, k)$ — оператор.

Оператор S_1 является *оператором заголовка процедуры,* имеющим в простейшем случае вид

m: procedure main; или m:процедура главная;

где m — метка (обязательно единственная в этом операторе), для ключевых слов **PROCEDURE** или **ПРОЦЕДУРА** допустимо сокращение **PROC** или **ПРОЦ**. Оператор S_k является *оператором конца*; в простейшем случае тело этого оператора состоит только из ключевого слова **END** или **КОНЕЦ**. Например, формально допустима следующая программа, не задающая выполнения каких-либо действий:

с английскими словами	с русскими словами
null: procedure main; end;	пусто:процедура главная; конец;

Проверь себя

1. Каков общий вид оператора PL/1-программы?
2. Какими операторами должна начинаться и заканчиваться PL/1-программа?
3. Допустимы ли операторы заголовка процедуры, не имеющие меток?

2.2. Оператор описания

Для явного указания свойств переменных используется *оператор описания.* В простейшем случае этот оператор имеет вид

declare $d_1\ d_2 \ldots d_m$ $(m \geq 1)$ или описание $d_1\ d_2 \ldots d_m$ $(m \geq 1)$.

Здесь d_i $(i = 1, 2, \ldots m)$ — описание свойств одной переменной, имеющее вид $ua_{г}a_{г}. \ldots a_k$ $(k \geq 0)$, где u — имя переменной; $a_1 \ldots a_к$ — ее описатели. Ключевое слово DECLARE или ОПИСАНИЕ может быть записано в сокращенной форме — DCL или ОПС.

Для описания характеристик значений числовых переменных применяются описатели:

а) *типа* — **REAL** или **COMPLEX** (вещественный или комплексный тип соответственно), в рассматриваемой версии REAL явно не указывается;

б) *формы представления* — **FIXED** или **FLOAT** (с фиксированной или с плавающей точкой соответственно);

в) *основания* (системы счисления) — **DECIMAL** или **BINARY** (десятичное или двоичное основание соответственно);

г) *точности* — (p) или (p, q), где p и q— целые положительные числа (см. § 1.5).

Русские эквиваленты английских слов **КОМПЛЕКСНОЕ, ТОЧНОЕ, ВЕЩЕСТВЕННОЕ, ДЕСЯТИЧНОЕ** и **ДВОИЧНОЕ** соответственно.

Описатель точности (p) применяется, если значения переменной представлены в форме с плавающей точкой, а описатель точности (p, q) — если значения представлены в форме с фиксированной точкой. Однако если $q = 0$, то описатель $(p, 0)$ может быть представлен просто в виде (p). Максимальные допустимые значения параметра p описателя точности приведены в табл. 2.1, значение параметра q должно лежать в диапазоне от 0 до 15.

ТАБЛИЦА 2.1.

Максимальные значения параметра p описателя точности

Форма представления	Основание	
	Двоичное	Десятичное
С фиксированной точкой	63	15
С плавающей точкой	53	16

Для ключевых слов COMPLEX, DECIMAL, BINARY, ВЕЩЕСТВЕННОЕ допустимы, соответственно, сокращения **CPLX, DEC, BIN, ВЕЩ**.

Все описатели числовой переменной, кроме описателя точности, могут указываться в операторе описания в произвольном порядке. Описатель точности должен следовать непосредственно за любым из описателей: FLOAT, FIXED, DECIMAL, BINARY, ВЕЩЕСТВЕННОЕ, ТОЧНОЕ, ДЕСЯТИЧНОЕ, ДВОИЧНОЕ. Например, переменную *x*, принимающую вещественные значения в форме с фиксированной точкой с десятичным основанием и точностью (5, 2), можно описать любым из следующих операторов описания:

1) dcl x fixed (5, 2) decimal;
2) опс x точное(5, 2) десятичное;
3) dcl x fixed decimal (5, 2);

Однако следующий оператор для описания переменной X неприемлем: DCL X (5, 2) FIXED DECIMAL; он будет воспринят компилятором как описание двумерного массива.

В описании свойств переменной можно опускать какие-либо описатели в операторе описания. В этом случае действуют правила, называемые *правилами по умолчанию,* существо которых заключается в следующем.

1. Если не задан описатель COMPLEX или КОМПЛЕКСНОЕ, то переменной автоматически приписывается описатель REAL, однако явно REAL не указывается.
2. Если опущен описатель формы представления (FLOAT/ВЕЩЕСТВЕННОЕ или FIXED/ТОЧНОЕ), но присутствует хотя бы один из описателей DECIMAL/ДЕСЯТИЧНОЕ или BINARY/ДВОИЧНОЕ, то при наличии описателя точности *(p, q)* переменной приписывается описатель FIXED/ТОЧНОЕ, иначе — описатель FLOAT/ВЕЩЕСТВЕННОЕ.
3. Если опущен описатель основания (DECIMAL/ДЕСЯТИЧНОЕ или BINARY/ДВОИЧНОЕ), но присутствует хотя бы один из описателей COMPLEX/КОМПЛЕКСНОЕ, FIXED/ТОЧНОЕ или FLOAT/ ВЕЩЕСТВЕННОЕ, то переменной приписывается описатель BINARY/ДВОИЧНОЕ.
4. Если опущен описатель точности, то он устанавливается в зависимости от других описателей, заданных явно или по умолчанию в описании переменной. Так, если заданы описатели FIXED/ТОЧНОЕ и DECIMAL/ДЕСЯТИЧНОЕ, то в качестве описателя точности выбирается описатель (6, 0); если заданы описатели FIXED/ТОЧНОЕ и BINARY/ДВОИЧНОЕ, то автоматически выбирается описатель точности (15,0); если заданы описатели FLOAT/ВЕЩЕСТВЕННОЕ и DECIMAL/ДЕСЯТИЧНОЕ, то в качестве описателя точности выбирается описатель (6); если заданы описатели FLOAT/ВЕЩЕСТВЕННОЕ и

BINARY/ДВОИЧНОЕ, то в качестве описателя точности выбирается описатель (24).

5. Если в описании переменной опущены все описатели типа ее значения, то выдается предупреждение и назначаются описатели FIXED/ТОЧНОЕ BINARY/ ДВОИЧНОЕ с описателем точности (15,0).

6. Если переменная вообще не описана в операторах описания, то выдается сообщение об ошибке.

В тех случаях, когда в одном операторе описания требуется описать несколько переменных, непосредственно следующих друг за другом, у которых все или часть описателей совпадают, можно сократить запись оператора описания, вынося справа за скобки описатели, общие для всех имен, заключенных в скобки. Так, например, оператор

```
dcl A float(53), B float(53) CPLX, C float(53) cplx, D float(53) cplx, E float(53);
```

можно представить в виде

```
dcl (A, B cplx, C cplx, D cplx, E) float (53);
```

Часть описаний, заключенных в скобки, может быть в свою очередь снова заключена в скобки с вынесением описателей, общих только для этой части переменных. Так, приведенный выше оператор может быть записан в еще более краткой форме;

```
dcl (A, (B, C, D) cplx, E) float (53);
```

Для хранения в памяти машины значения переменной отводится разное количество байтов в зависимости от состава описателей, указанных в описании переменной. Эта зависимость отражена в табл. 2.2.

ТАБЛИЦА 2.2.

Состав описателей, заданных при описании свойств переменной	Количество байтов памяти
DECIMAL FIXED(p, q)	p/2+1 при четном p (p+1)/2 при нечетном p
BINARY FIXED(p)	1 при $p \le 7$ 2 при $p > 7$ $p \le 15$ 4 при $p > 15$ и $p \le 31$ 8 при $p > 31$
DECIMAL FLOAT(p)	4 при $p \le 6$ 8 при $p > 6$
BINARY FLOAT(p)	4 при $p \le 24$ 8 при $p > 24$

Для хранения значений переменной комплексного типа требуется ровно в два раза больше памяти, чем для хранения значений переменной вещественного типа с той же формой представления и точностью.

Проверь себя

1. Какие средства в PL/1-программе применяются для описания свойств (характеристик) переменных?
2. Указать диапазон допустимых значений *p* и *q* в описателе точности для значений переменных, представленных в форме с фиксированной точкой *а)* в двоичной системе счисления, б) в десятичной системе счисления.
3. Указать, какие из следующих чисел: 200; -1,50; 2,33; -4040; 1,00; 0,01; 0,050; 0,10; 6 могут быть точными значениями переменной с вещественными десятичными значениями в форме с фиксированной точкой, если описатель точности переменной имеет вид

1) (2, 1)
2) (3)
3) (3,1)
4) (2, 2) .
5) (5, 2)
6) (8, 2)

4. Любое ли число в форме с плавающей точкой, входящее в допустимый диапазон значений некоторой переменной, представимо в памяти без потери точности?
5. Какие описатели предполагаются по умолчанию в приведенных описаниях числовых переменных:

1) dcl A fixed(7);
2) опс A двоичное(31);
3) опс A вещ (24);
4) dcl A ninary(15);
5) dcl A complex float;
6) dcl A binary fixed;
7) dcl A fixed;
8) dcl A binary;
9) dcl A float;
10) dcl A;

6. Переписать приведенный оператор описания переменных в наиболее краткой форме, опустив все предполагаемые по умолчанию описатели и применив вынесение описателей за скобки:

```
declare
a decimal fixed (15),
в fixed,
c fixed binary(15),
d binary fixed(15),
e complex float(24);
```

7. Значения какой переменной занимают меньший объем памяти: типа fixed decimal(4, 0) или типа fixed binary(15, 0)?

2.3. Оператор присваивания

Оператор присваивания предназначен для присваивания переменной или совокупности переменных значения выражения. В простейшем случае этот оператор имеет вид

$$u=e;$$

где u — имя переменной; e — выражение, значение которого присваивается переменной v. Например, если a, b и c — имена переменных, то следующие конструкции языка будут операторами присваивания:

a=1; (установить значение переменной a, равное 1),

b=b+2; (увеличить на 2 значение переменной b),

c=a*b; (присвоить переменной c значение, равное произведению значений переменных a и b).

Часть оператора присваивания, находящуюся слева от знака равенства, принято называть его левой частью, а часть оператора, стоящую справа от знака равенства — правой частью.

Выполнение оператора присваивания сводится к вычислению значения выражения e с последующим присвоением вычисленного значения выражения e переменной u, если характеристики значений выражения e и переменной u совпадают. Однако тип значения выражения e может не совпадать с типом допустимых значений переменной u. В этом случае значение выражения e автоматически преобразуется к типу, требуемому для переменной u. Так же, как и при преобразовании операндов операции, здесь при уменьшении точности значения лишние младшие цифры исходного значения отбрасываются без какого-либо округления. Например, если A — переменная с вещественными значениями с фиксированной точкой и точностью (15, 0), то после выполнения оператора A= 13.999; значение переменной A станет равным 13. Учитывая эту особенность языка PL/1, следует с осторожностью использовать операторы присваивания, в левой части которых указана переменная, принимающая значение в форме с фиксированной точкой, а в правой части — выражение, принимающее значение в форме с плавающей точкой. Например, если B — переменная, принимающая значение в форме с плавающей точкой, а C — переменная, принимающая значение с десятичным основанием, в форме с фиксированной точкой и точностью (15, 5), то после выполнения операторов

```
b=1e-5; c=b;
```

переменной C будет присвоено нулевое значение. Это связано с тем, что представление числа 1E—5 в форме с плавающей точкой не является точным и фактически меньше, чем 0,00001.

Если в левой части оператора присваивания указана переменная с комплексными значениями, а в правой части оператора стоит выражение с действительным числовым значением, то переменной u присваивается значение с нулевой мнимой частью.

Если переменная u принимает действительные значения, а выражение e — комплексные, то сообщается об ошибке.

Значение выражения в правой части оператора присваивания может оказаться за пределами, допустимыми для переменной u, указанной в левой части оператора. Для переменных, принимающих значение в форме с плавающей точкой, это приводит к аварийному завершению выполнения программы (но это можно отменить или перехватить средствами языка). Для переменных, принимающих значение в форме с фиксированной точкой и точностью (p, q), возможны в этом случае две ситуации. Если значение выражения не может быть представлено при данном основании с точностью (N, q), где N равно 15 для десятичных чисел, то программа аварийно завершится (и это тоже можно перехватить/отменить средствами языка). Если же значение выражения может быть представлено с точностью (N, q), но не может быть представлено с точностью (p, q), то результат присваивания, вообще говоря, не определен, однако при этом прерывание выполнения программы не произойдет, если не использованы специальные средства языка, описанные в гл. 8.

Другой тип оператора присваивания имеет вид

$$u_1, u_2, \ldots, u_n = e;$$

Описатели типа переменных $u_1, u_2, \ldots, u_n$ могут не совпадать. При выполнении этого типа оператора присваивания производятся следующие действия:

1) вычисляется значение выражения e, содержащегося в правой части оператора;

2) значение правой части оператора преобразуется в соответствии с описателями переменной u_i $(i=1, 2, \ldots, n)$, если описатели этой переменной не совпадают с характеристиками значения выражения e, с последующим присвоением его переменной u_i.

Обратите внимание на тонкий момент: если и слева в списке переменных и справа в выражении имеется одна и та же переменная, например, x, то если она стоит первой в списке слева, то она же и первая изменится, а затем все остальные переменные получат значения выражения с уже измененной переменной x. Если переменная x стоит слева в списке последней, то остальные переменные получат значения выражения с исходным значением x, и только потом и сама переменная x получит новое значение.

Поскольку в программах часто встречаются операторы присваивания вида X=X+1; т.е. правая часть начинается с повторения левой части, для сокращения записи таких операторов используется форма $M\ n\ =\ e$, где M – имя переменной в левой части присваивания, n – одна из следующих операций:

+ (плюс), - (минус), * (умножить), / (делить), | (логическое ИЛИ), & (логическое И), || (склейка строк), ** (возведение в степень)

e – выражение правой части присваивания без переменной M и знака операции n. Примеры.

X+=1; S1||=S2; Y**=2-Z; B1 &= B2;

Проверь себя

1. Для чего предназначен оператор присваивания?
2. Должен ли тип значения выражения в правой части оператора присваивания совпадать с типом значения переменной в левой части этого оператора?
3. Пусть переменная A имеет вещественные десятичные значения в форме с фиксированной точкой с точностью (5, 0). Каково будет значение переменной A после выполнения оператора A= 0.99999;
4. Пусть характеристики переменных A, B и C описаны в операторе

   declare ((A, B) complex, C) float;

каковы будут значения переменных B и C после выполнения операторов A='3+5i'; C=real(A); B=C;

5. Пусть характеристики переменных A, B и C описаны в операторе

   опс A десятичное (15, 10), B десятичное (5, 0), C десятичное (15);

к каким последствиям ведет выполнение следующих операторов;

   а) C= 999999; A=C; б) C= 999999; B=C;

6. Составить один оператор присваивания, эквивалентный последовательности операторов: X=2; Y=2; N=2.
7. Написать в сокращенной форме операторы

   X=X+Y; Z=Z/2; B1=B1 & B2 & B3; X=X**2;

2.4. Ввод и вывод данных

Результаты, получаемые в процессе решения задачи, практически всегда передаются из оперативной памяти на какие-либо внешние устройства, например, на печатающие, на диски или в сеть. Процесс передачи данных из оперативной памяти на внешние носители называется *выводом данных.* Исходные данные для решения задачи передаются в оперативную память с каких-либо внешних носителей: дисков или из сети. Этот процесс называется *вводом данных.* Процесс ввода-вывода данных осуществляется с помощью операторов ввода и вывода данных. В языке PL/1 предусмотрены разнообразные операторы для выполнения ввода и вывода данных. Подробно они рассматриваются в гл. 6, 10, 13. Однако, учитывая необходимость ввода и вывода данных даже в программах с простейшей структурой, в настоящей главе рассмотрим наиболее часто используемые формы операторов ввода-вывода.

Простейшие *операторы ввода* (GET/ЧИТАТЬ) и *вывода* (PUT/ПИСАТЬ) данных имеют вид

С английскими словами	С русскими словами
get list(s); put list(s);	читать в_виде(s); писать в_виде(s);

а в самом простейшем случае даже get (s); put (s); или читать (s); писать (s); Вместо ключевого слова ПИСАТЬ в данной версии языка разрешено использовать и слово ПЕЧАТАТЬ.

Здесь s — список данных вида $(a_1, a_2, ..., a_n)$ $(n \geq 1)$, где a_i — элемент списка данных. В случае оператора ввода элементами списка данных могут быть, в частности, имена переменных. Например, оператор

get list (x1, y, a, b); или читать в_виде(x1, y, a, b);

задает ввод значений переменных $x1, y, a, b$.

Вводимые значения должны следовать в порядке, строго соответствующем порядку элементов списка данных. Вводимые числовые значения представляются в форме любых числовых констант, допустимых в языке PL/1. Для комплексных значений должна вводиться пара констант, соответствующих действительной и мнимой частям значения.

На внешних носителях для отделения вводимых значений друг от друга достаточно поместить между ними один или несколько пробелов или запятую.

Последовательность символов, изображающих вводимые значения, называемая *входным потоком,* всегда рассматривается как непрерывная.

Тип константы, изображающей вводимое значение, не обязан совпадать с типом значений соответствующей переменной. При необходимости вводимое

значение будет преобразовано к требуемому типу так же, как и в случае операторов присваивания (см. § 2.2).

В случае оператора вывода (PUT/ПИСАТЬ) элементами списка данных могут быть любые выражения. Например, оператор

```
put list (X, 5, Y**2-3);          писать в_виде (X, 5, Y**2-3);
```

задает вывод значения переменной X, числа 5 и значения выражения y^2-3. Порядок, в котором выводятся значения, всегда соответствует порядку элементов списка данных. Выводимые вещественные числа представляются в виде десятичных констант, комплексные — в виде пары таких же констант, автоматически разделяемых знаком плюс-минус и оканчивающимися символом «I». При печати данных каждое очередное значение обычно размещается в строке, пока хватает места. Кроме того, наглядность выводимого текста можно повысить, используя операторы вида:

```
put list(s) skip;                 писать в_виде(s) с_новой;
put skip list (s);                писать с_новой в_виде(s);
```

Эти операторы отличаются от рассмотренных ранее тем, что при их выполнении вывод значений всегда начинается с новой строки (*s* в данном случае, как и выше, список выводимых данных).

Наглядность выводимых данных всегда улучшается, если их сопровождать текстовыми комментариями, указывающими, какие значения выводятся. Применяемые для этого средства языка будут рассмотрены в последующих главах. Здесь же укажем еще одну форму операторов вывода, при выполнении которых данные всегда выводятся в виде *u*=*c*, где *u* — имя переменной, значение которой выводится; *c* — константа, изображающая выводимое значение. Эти операторы вывода имеют вид

```
put data(s);                      писать с_именами(s);
put skip data(s);                 писать с_новой с_именами(s);
put data(s) skip;                 писать с_именами(s) с_новой;
put data(skip,s);                 писать с_именами(с_новой, s);
```

где *s* — список данных, элементами которого должны быть имена переменных. Ключевое слово **SKIP/С_НОВОЙ** и в данном случае означает, что при выполнении оператора вывода данные всегда будут выводиться с новой строки. Например, после выполнения операторов:

```
dcl(x, y, al, bl, c1) fixed(l);   опс (x, y, al, bl, c1) точное(1);
x, y, a1, в1, c1 = 1;             x, y, a1, в1, c1 = 1;
put skip data(x, y);              писать с_новой с_именами(x,y);
put skip data(a1, bl, c1);        писать с_новой с_именами(a1, bl, c1);
```

будет напечатано

```
X=  1 Y=  1
```

```
A1=  1 B1=  1 C1=  1
```

Операторы **GET/ЧИТАТЬ** или **PUT/ПИСАТЬ** с конструкцией **LIST/В_ВИДЕ** называют *операторами ввода* или *вывода списка данных,* а операторы с конструкцией **DATA/С_ИМЕНАМИ** — *операторами ввода* или *вывода именованных данных*. В рассматриваемой версии языка операторы GET DATA или ЧИТАТЬ С_ИМЕНАМИ реализованы в упрощенном виде, как операторы «зеркальные» по отношению к операторам PUT DATA или ПИСАТЬ С_ИМЕНАМИ.

Проверь себя

1. Что называется вводом и выводом данных?
2. Каков вид простейших операторов ввода и вывода?
3. Составить входной поток для ввода значений переменных Al, Bl, X, Y, Z, если эти значения соответственно равны

-1/5, $66 \cdot 10^{7}$, $3 \cdot 2^{-13}$, 2i, -54 -7i,

а оператор ввода имеет вид

get list(x, y, z, al, bl); читать_в_виде(x, y, z, al, bl);

4. Какие числа будут напечатаны после выполнения операторов

```
N = -5; M, K= 18;
Писать в_виде(M, N + K, M*K + N-100);
```

при условии, что все переменные имеют двоичные вещественные значения с фиксированной точкой и точностью (15, 0)?

5. Что означает ключевое слово SKIP/С_НОВОЙ в операторе PUT/ПИСАТЬ?
6. В каком виде выводятся данные посредством операторов PUT/ПИСАТЬ с ключевым словом DATA/С_ИМЕНАМИ?

2.5. Порядок выполнения операторов. Условный оператор

Выполнение процедуры обычно начинается с ее первого выполняемого оператора. При отсутствии специальных указаний о передаче управления операторы будут выполняться последовательно, один за другим в соответствии с их расположением в тексте процедуры. Такой порядок выполнения операторов принято называть естественным. Операторы описания могут быть расположены в любом месте программы с простейшей структурой. Если эти операторы встречаются в ходе выполнения программы, то они пропускаются. Например, в программе:

```
example:proc main;                пример:проц главная;
get list(x,y);                    читать в_виде(x,y);
```

```
z=sqrt(x**2-y**2);
dcl (x, y, z) float(53);
put list(z);
end;
```

```
z=sqrt(x**2-y**2);
опс (x,y,z) вещ(53);
писать в_виде(z);
конец;
```

первым будет выполнен оператор ввода, затем управление будет передано оператору присваивания и последним будет выполнен оператор вывода.

Одним из наиболее часто применяющихся операторов, которые изменяют естественный порядок выполнения операторов программы, является *условный оператор,* имеющий одну из следующих форм:

1) if *e* then t_1 — если *e* тогда t_1
2) if *e* then t_1 else t_2 — если *e* тогда t_1 иначе t_2

Здесь e — выражение; t_1 и t_2 — либо отдельный оператор (кроме операторов DECLARE/ОПИСАНИЕ, DO/ЦИКЛ, PROCEDURE/ ПРОЦЕДУРА, BEGIN/БЛОК, END/КОНЕЦ, FORMAT/ВВЕСТИ_ФОРМАТ), либо *группа операторов* (см. ниже), либо *блок* (см. гл. 4).

Каждая из конструкций t_1 и t_2 должна заканчиваться точкой с запятой. Выражение e принято называть *логическим.* Обычно в качестве логических используются выражения, значения которых являются строками из одного бита (напомним, что результаты всех операций сравнения являются строками из одного бита). В этом случае условный оператор выполняется следующим образом:

- вычисляется значение выражения *e;*
- проверяется условие, равно ли это значение биту 1;
- если равно, то управление передается t_1; после выполнения (если не изменится порядок выполнения программы) управление передается оператору, следующему за t_2 (если конструкция ELSE/ИНАЧЕ задана) либо за t_1 (если конструкция ELSE/ИНАЧЕ не задана);
- если не равно, то управление передается t_2 (если конструкция ELSE/ИНАЧЕ задана) либо оператору, следующему за t_1 (если конструкция ELSE/ИНАЧЕ не задана).

Правила выполнения условных операторов в тех случаях, когда значение выражения e не является строкой из одного бита, рассмотрены в гл. 5.

Рассмотрим примеры условных операторов. Вычисление функции

$$f(x)=\begin{cases}\sqrt{x} & \text{при } x>0\\ 0 & \text{при } x\le 0\end{cases}$$

может быть задано с помощью операторов

```
if x>0 then fx=sqrt(x); else fx = 0;      если x>0 тогда fx=sqrt(x) иначе fx=0;
```

Вычисление функции

$$g(x)=\begin{cases} 0 \text{ при } 0 \le x \le 2 \\ x^2 - 2x \text{ в остальных случаях} \end{cases}$$

может быть задано с помощью операторов

```
if x>=0 & x<=2 then gx =0;              если x>=0 & x<=2 тогда gx =0;
        else gx = x**2-2*x;                          иначе gx = x**2-2*x;
```

Оператор, входящий в состав условного, сам также может быть условным. При этом в последовательности операторов вида:

if e_1 then if e_2 then t_1 else t_2 если e_1 тогда если e_2 тогда t_1 иначе t_2

конструкция ELSE/ИНАЧЕ относится к внутреннему условному оператору. В качестве примера рассмотрим программу, вычисляющую значение функции

$$f(x)=\begin{cases} g1(x) & \text{при } x < 0 \\ g2(x) & \text{при } x \ge 0 \end{cases}$$

$$g_1(x)=\begin{cases} 2x+1 \text{ при } x > -0{,}5 \\ -2x-1 \text{ при } x \le -0{,}5 \end{cases}$$

$$g2(x)=\begin{cases} -x+1 \text{ при } x < 1 \\ x-1 \text{ при } x \ge 1 \end{cases}$$

Программа может иметь вид

```
gfx: proc main;                          gfx: проц главная;
get list(x);                             читать в_виде(x);
if x <0 then                             если x <0 тогда
if x > -0.5 then fx = 2*x + 1;           если x > -0.5 тогда fx =  2*x + 1;
            else  fx =-2*x - 1;                      иначе fx =-2*x - 1;
        else                                     иначе
         if x< 1 then fx = - x + l;               если x< 1 тогда fx = - x + l;
                 else fx=     x - 1;                          иначе fx=    x - 1;
put list(fx);                            писать в_виде(fx);
end;                                     конец;
```

Другой пример. Вычисление величины SIGN, равной 1 (при $x > 0$), -1 (при $x < 0$) или 0 (при $x = 0$) может быть задано последовательностью операторов:

```
if x>0 then sign =l;                    если x>0 тогда sign =l;
       else if x<0 then sign = -1;              иначе если x<0 тогда sign = -1;
                   else  sign = 0;                          иначе sign = 0;
```

В тех случаях, когда действия, подлежащие выполнению при определенном условии, не могут быть записаны одним оператором, удобно применять группы операторов. По определению *группа операторов* есть последовательность, начинающаяся с оператора начала группы и

заканчивающаяся оператором конца. Оператор начала группы имеет ключевое слово **DO** («выполнить») или символа **{** и в простейшем случае состоит только из этого ключевого слова. Более сложные формы операторов DO/ЦИКЛ применяются в *циклических группах* или *циклах* (см. §3.1).

Рассмотрим пример; пусть в случае, когда значение переменной x1 меньше значения переменной x2, необходимо «поменять местами» значения этих переменных (т. е. значение переменной x1 присвоить переменной x2, а значение переменной x2 — переменной x1). С использованием дополнительной переменной y эти действия могут быть заданы операторами:

```
if x1 < x2 then                    если x1 < x2 тогда
do; y=x2; x2= x1; x1=y; end;       {; y=x2; x2= x1; x1=y; };
```

Отметим, что для таких действий в рассматриваемой версии языка существует оператор *обмена*, который «обменивает» переменные значениями одним действием. Этот оператор записывается тремя символами <=>. И приведенный выше пример можно записать без использования группы

```
if x1 < x2 then x1<=>x2;          если x1 < x2 тогда x1 <=> x2;
```

В качестве более сложного примера рассмотрим программу, которая вычисляет координаты какой-либо точки, принадлежащей прямой, заданной уравнением a*x*+b*y*+c= 0 (|a|+|b| > 0).

Программа может иметь вид

```
point:proc main;                   точка:проц главная;
get list(a, в, c);                 читать в_виде(a, в, c);
if a^=0 then do; x = -c/a; y = 0; end;   если a^=0 тогда  {; x = -c/a; y = 0; };
        else  do; y = -c/b; x = 0; end;          иначе {; y = -c/b; x = 0; };
put list(x, y);                    писать в_виде(x, y);
end;                               конец;
```

Выше были рассмотрены различные случаи применения операторов конца (**END/КОНЕЦ**): для указания конца текста процедуры, конца группы операторов. Последовательности операторов, ограничиваемые оператором конца, могут входить одна в другую, но только целиком, а не по частям. В конце каждой подобной последовательности при этом должен стоять ее собственный оператор END/КОНЕЦ. В силу этого требования часто возникают случаи, когда необходимо записать подряд несколько операторов конца. Например, программа вычисления и печати значения функции

$$f(x)=\begin{cases}0 & \text{при } x < 0\\ sin^2 - 2\sin x & \text{при } x \geq 0\end{cases}$$

имеющая вид

```
fx:proc main;                      fx:проц главная;
get list(x);                       читать в_виде(x);
```

```
if x <0 then put list(0);
else do;
y=sin(x);
put list(y**2—2*y);
end;
end;
```

```
если x <0 тогда писать в_виде(0);
иначе цикл;
y=sin(x);
писать в_виде(y**2—2*y);
конец;
конец;
```

содержит два оператора end/конец, непосредственно следующих друг за другом. Первый оператор конца замыкает группу, начинающуюся с оператора DO/ЦИКЛ; второй — замыкает всю процедуру. В языке PL/1 имеется возможность обозначить отличия операторов END/КОНЕЦ друг от друга. После слова END/КОНЕЦ, но до точки с запятой можно повторить метку соответствующей процедуры или параметр цикла или метку, специально поставленную для целей контроля перед началом группы. Например, в приведенном выше примере можно закончить текст оператором end fx; или конец fs; и компилятор проверит, что это действительно оператор, оканчивающий весь текст процедуры fs.

В рассматриваемой версии языка имеется еще одна форма оператора группы DO/**{**. Если перед ключевым словом DO/**{**; поставить префикс из цифры **1**, то последовательность операторов группы будет выполнено только один раз, даже, если управление попадает на это место в программе несколько раз, например: 1 {; a=10; b=5; }; здесь присваивание переменным a и b произойдет только при первом выполнении этой группы. Больше во время работы программы управление внутрь этой группы не попадет ни разу.

Проверь себя

1. В каком порядке выполняются операторы при отсутствии специальных указаний о передаче управления?
2. Из каких частей состоит составной оператор IF/ЕСЛИ?
3. Что будет напечатано после выполнения операторов

```
a = 3;
if a < 2 ! a ^> 4 then
   put list(l);
          else
   put list(2);
```

```
a = 3;
если a < 2 ! a ^> 4 тогда
      писать в_виде(l);
             иначе
      писать в_виде(2);
```

4. Составить условный оператор, позволяющий вычислить значение функции *f(x)*, заданной следующей формулой:

$$f(x)=\begin{cases}\sin x & \text{при } x > 0\\ 1-\cos x & \text{при } x \le 0\end{cases}$$

$$f(x)=\begin{cases} 0 & \text{при } -2 \le x \le 0 \\ x^2 - 2x & \text{в остальных случаях} \end{cases}$$

5. Используя вложенные условные операторы, составьте программу, вычисляющую и печатающую значение функции *g(x, y)*, заданной формулой:

$$g(x,y)=\begin{cases} x+y & \text{при } x>1,\ y>1 \\ x-y & \text{при } x>1,\ y\le 1 \\ -x+y & \text{при } x\le 1,\ y>0 \\ -x-y & \text{при } x\le 1,\ y\le 0 \end{cases}$$

6. Привести пример использования группы операторов, ограниченной операторами

DO; ...; END; или {; };

2.6. Оператор перехода

Оператор перехода имеет вид

1) **go to m** 2) **goto m** 3) **идти m**

где m — метка какого-либо оператора (кроме операторов PROCEDURE/ ПРОЦЕДУРА, DECLARE/ОПИСАНИЕ, FORMAT/ ВВЕСТИ_ФОРМАТ) либо имя переменной типа метки (см. ниже). Если m — метка, то после выполнения оператора перехода следующим будет выполняться оператор с меткой m. Например, данные две последовательности операторов функционально эквивалентны:

if *e* then do;	if ^(*e*) then goto m;	если *e* тогда {;	если ^(*e*) тогда идти m;
s_1	s_1	s_1	s_1
s_2	s_2	s_2	s_2
…	…	…	…
s_n	s_n	s_n	s_n
end;	m:;	};	m:;

Переменные типа метки относятся к переменным управляющего типа; они принимают значения, являющиеся метками операторов (кроме меток операторов PROCEDURE/ПРОЦЕДУРА). Переменные типа метки должны быть явно описаны в операторе DECLARE/ОПИСАНИЕ. При этом им приписывается описатель вида **LABEL/МЕТКА**.

Для установки значений переменных типа метки может быть использован обычный оператор присваивания вида

$$u_1\ u_2, \ldots, u_n = e\ (n \ge 1),$$

где u_i $(i=1, 2,\ldots,n)$ — имя переменной типа метки; e — либо метка оператора, либо имя другой переменной типа метки.

После выполнения оператора перехода GOTO/ИДТИ *u*, где *u* — имя переменной типа метки, следующим будет выполняться оператор с меткой, равной значению переменной *u*. Например, для вычисления и печати значения выражения, используя только операторы перехода

$$\sum_{i=1}^{k} x^i + \sum_{i=1}^{n} y^i \quad (k \geq 2, n \geq 2)$$

может быть использована следующая программа:

```
psum: proc main;
dcl l label, (x,y)(100) float,
(k, i, n) fixed(31);

get list (x, y, k, n);
z = x;
m= k;
l=m1;
goto cs;
ml: s1=s;
      z = y;
      m=n;
      l=m2;
      goto cs;
m2: put list(s1 + s);
      goto me;
cs:  i=2;
      s=z;
mc: s = s + z**i;
if i =m then goto l;
i = i + 1;
goto mc;
me: end;
```

```
psum: проц главная;
опс l метка, (x,y)(100) вещ,
(k, i, n) точное(31);

читать в_виде (x, y, k, n);
z = x;
m= k;
l=m1;
идти cs;
ml: s1=s;
      z = y;
      m=n;
      l=m2;
      идти cs;
m2: писать в_виде(s1 + s);
      идти me;
cs:  i=2;
      s=z;
mc: s = s + z**i;
если i =m тогда идти l;
i = i + 1;
идти mc;
me: конец;
```

Здесь при первом выполнении оператора GOTO/ИДТИ L; управление будет передано оператору с меткой Ml, а при втором выполнении — оператору с меткой M2.

В языке PL/l к переменным типа метки применимы операции сравнения «равно» (=) и «не равно» (^=).

При выполнении тех или иных операторов может возникнуть ситуация, когда естественный порядок выполнения оператора невозможен, например, при делении на нуль или при превышении допустимого максимума результата

операции. При этом выполнение оператора прерывается и осуществляется обработка возникшей ситуации. Стандартное действие при этом обычно состоит в печати соответствующего сообщения и завершении выполнения программы. Однако в языке PL/1 программисту предоставлена возможность самому задать действия, которые должны быть выполнены при возникновении определенной ситуации. Подробно соответствующие средства языка будут изложены в гл. 8. Здесь же рассмотрим лишь один из случаев нестандартной обработки прерываний, а именно для ситуации, возникающей, когда при выполнении оператора ввода оказывается, что данных во входном потоке уже нет. В простейшем случае обработка подобного прерывания может быть задана оператором вида

```
on endfile (sysin) goto m;          когда конец_файла(стд_ввод) идти m;
```

где m — метка оператора, которому должно быть передано управление по достижении конца вводимых данных. В качестве примера использования такого оператора приведем программу, вычисляющую среднее арифметическое произвольной последовательности чисел:

```
gsa: proc main;                     gsa: проц главная;
dcl (as,x) float, n fixed;          опс (as,x) вещ, n точное;
on endfile(sysin) goto me;          когда конец_файла(стд_ввод) идти me;
as, n = 0;                          as, n = 0;
m: get list(x);                     m: читать в_виде(x);
as=as+ x;                           as=as+ x;
n+=1;                               n+=1;
goto m;                             идти m;
me: put list(as/n);                 me: писать в_виде(as/n);
end;                                end;
```

Оператор, задающий обработку прерывания (**ON/КОГДА**), должен быть выполнен до того, как возникает ситуация, требующая обработки. В рассмотренном примере такой оператор должен быть выполнен раньше оператора ввода.

Проверь себя

1. Чему будет равно значение переменной x после выполнения фрагмента программы:

```
x = 3;
goto м;
x = l;
м: ;
```

2. Метки каких операторов могут быть значениями переменных типа метки?

3. Каково назначение и общий вид описателя LABEL/МЕТКА?

4. Используя оператор перехода вида GOTO *u* или ИДТИ *u*, где u — имя переменной типа метки, составьте программу для вычисления выражения $\prod_{j=1}^{k} \sin(\pi j x) + \prod_{j=1}^{m} \sin(\pi j y)$.

5. С какой целью используется оператор вида

on endf1le (sysin) goto m; когда конец_файла(стд_ввод) идти m;

6. Составьте программу, вычисляющую среднее геометрическое произвольной последовательности чисел (при условии, что количество чисел в этой последовательности во входном потоке не указано).

7. В чем состоит ошибка в следующей программе:

```
why: proc main;                    почему: проц главная;
dcl (x,y,z) float;                 опс (x,y,z) вещ;
м: get list(x, y, z);              м: читать в_виде(x, y, z);
put list (x*y*z);                  писать в_виде (x*y*z);
goto m;                            идти м;
on endfile(sysin) goto me;         когда конец_файла(стд_ввод) идти me;
me: end;                           me: end;
```

2.7. Простейшая форма заданий на трансляцию и выполнение PL/1-программ

Для того чтобы оттранслировать и выполнить программу на языке PL/1, с помощью бесплатного компилятора, представленного на сайте pl1.su, требуется запустить из командной строки Windows файл PLINK64.EXE, указав ему имя файла с текстом PL/1-программы.

Поскольку файл PLINK64.EXE содержит не только компилятор с языка PL/1, но и ряд утилит, необходимых для работы в среде PL/1, требуется указать и расширение файла с исходным текстом (в кодировке «кириллица DOS»), которое должно быть или .PL1 или .PLI.

Подробно работа с компилятором изложена в гл. 11.

В простейшем случае задание на трансляцию и последующее выполнение программы, составленной на языке PL/1, состоит из двух директив, заданных в командной строке Windows, и имеет вид

```
PLINK64 TEST.PL1
TEST
```

В качестве примера рассмотрим задание на трансляцию и выполнение программы, вычисляющей значение выражения x^9-15x^6+185:

```
test: proc main;                   test: проц главная;
```

```
dcl x float(53);                    опс x вещ(53);
get list(x);                        читать в_виде(x);
put list(x**9 - 15*x**6+185);       писать в_виде(x**9 - 15*x**6+185);
end;                                конец;
```

По умолчанию вывод идет на экран, а ввод – с клавиатуры.

После выполнения двух приведенных выше директив из командной строки, программа test.exe остановится и будет ждать ввода значения *x* с клавиатуры. Введя какое-либо значение (например, 155) и нажав Enter, пользователь заставляет завершиться оператор ввода, после чего будет напечатано 5.16396790238853E+019 т.е. значение выражения 155^9 -15*155^6 +185.

Проверь себя

1. Как запустить транслятор PL/1, доступный на сайте PL1.SU заданиями?
2. В какой кодировке Windows должен быть написан исходный текст PL/1-программы и какое расширение должен иметь этот файл?
3. Составьте программу, вычисляющую и печатающую значение выражения $\sin y^2 - \sin^2 y$; оттранслируйте ее и получите результат выполнения этой программы при $y = 12{,}85$.

3. ЦИКЛЫ. МАССИВЫ

3.1. Цикл. Организация цикла

В программе часто требуется организовать циклические повторения одних и тех же вычислений. Для этого удобно использовать циклические группы операторов. Они представляют собой последовательность операторов $s_0\ s_1\ \ldots\ s_n\ s_{n+1}$, где s_0 — оператор начала группы, а s_{n+1} — оператор конца (END/КОНЕЦ). Ранее были рассмотрены группы операторов, которые начинаются с оператора начала группы, состоящего из одного ключевого слова DO/ЦИКЛ или пары фигурных скобок, которые эквивалентны словам DO/ЦИКЛ и END/КОНЕЦ. Такие группы не являются циклическими. В качестве заголовка циклических групп операторов используются более сложные формы оператора **DO/ЦИКЛ**, которые можно разделить на два типа.

Оператор начала циклической группы *первого типа* имеет вид:

DO WHILE(*e*) или **ЦИКЛ ПОКА(*e*)**

Здесь e — логические выражение.

Процесс циклического выполнения группы операторов организуется в данном случае следующим образом. Перед тем как очередной (в том числе и первый) раз передать управление оператору вычисляется значение выражения e и если оно ложно, то цикл прекращается и управление передается оператору, непосредственно следующему за циклической группой.

Указанный процесс может быть представлен с помощью условных операторов и операторов перехода:

```
ml: if ^(e) then goto me;
s1
s2
sn
goto ml;
me: ;
```

```
ml: если ^(e) тогда идти me;
s1
s2
sn
идти ml;
me: ;
```

В качестве примера рассмотрим программы вычисления суммы

$$\sum_{n=1}^{N} x^n \quad (|x| < 1)$$

где N заранее известно и определяется либо из условий: $|x|^{N-1} > a$ и $|x|^N \le a$.

```
cs: proc main;
dcl (a,x,y,s) float(53);
get list(x, a);
y=x;
s=0;
do while(abs(y)>a);
```

```
cs: проц главная;
опс (a,x,y,s) вещ(53);
читать в_виде(x, a);
y=x;
s=0;
цикл пока(abs(y)>a);
```

```
    s+=y;                          s+=y;
    y*=x;                          y*=x;
end while;                     конец пока;
put list(s);                   put list(s);
end cs;                        конец cs;
```

Оператор начала циклической группы *второго типа* имеет вид

DO $u=p_1, p_2 \ldots p_m$ $(m \geq 1)$ или **ЦИКЛ** $u=p_1, p_2 \ldots p_m$ $(m \geq 1)$.

Здесь u — переменная любого типа, которая в данном контексте называется переменной, управляющей циклом; p_i $(i=1, 2, \ldots, m)$ — конструкция, имеющая вид $e^0_i\ e_i\ c_i$ и называемая спецификацией цикла. Здесь e^0_i — выражение, задающее начальное значение переменной u; e_i — либо пусто, либо конструкция, задающая изменение значений переменой u, c_i — либо пусто, либо конструкция, задающая условия выполнения цикла.

Каждая спецификация p_i, задает независимо от других спецификаций свой собственный порядок выполнения цикла. Как только цикл в соответствии со спецификацией p_i $(i=1,2,\ldots, m-1)$ будет прекращен, начнется выполнение цикла в соответствии со спецификацией p_{i+1} и так далее до тех пор, пока весь список спецификаций не будет исчерпан. Затем управление передается оператору, следующему непосредственно за оператором конца данной циклической группы. В каждой спецификации p_i может быть даже своя переменная u, управляющая циклом.

При переходе к каждой очередной спецификации во всех случаях вычисляется значение входящего в эту спецификацию выражения e^0_i, затем это значение присваивается переменной u.

Рассмотрим различные формы спецификаций цикла. Если конструкции e_i и c_i отсутствуют, то цикл, соответствующий этой спецификации, без каких-либо условий выполняется один раз.

Примером использования этой спецификации может служить программа, печатающая значения синусов углов 0,1, 0,25, 0,3, 0,4, 0,45:

```
psin: pr oc main;                  psin: проц главная;
do x = 0.1, 0.25, 0.3, 0.4, 0.45;  цикл x = 0.1, 0.25, 0.3, 0.4, 0.45;
      put list(sin(x));                  писать в_виде(sin(x));
end;                               конец;
end psin;                          конец psin;
```

Конструкция c_i может иметь одну из следующих форм:

1) **BY e^1_i** **С_ШАГОМ e^1_i**
2) **TO e^2_i** **ДО e^2_i**
3) **BY e^1_i TO e^2_i** **С_ШАГОМ e^1_i ДО e^2_i**
4) **TO e^2_i BY e^1_i** **ДО e^2_i С_ШАГОМ e^1_i**
5) **REPEAT (e^3_i)** **ПОВТОРЯЯ (e^3_i).**

Здесь значение выражения e^1_i равно приращению, прибавляемому к значению переменной u перед каждым очередным повторением цикла (значение e^1_i может быть отрицательным); значение выражения e^2_i определяет предел, до которого допускается изменять значение переменной u. Когда значение u выйдет за этот предел, выполнение цикла в соответствии с данной спецификацией будет прекращено. Значения выражений e^1_i и e^2_i вычисляются при переходе к спецификации p_i; и в дальнейшем не пересчитываются (они хранятся во вспомогательных переменных, обозначенных ниже через u_1 и u_2).

Конструкция REPEAT/ПОВТОРЯЯ(e^3_i) заставляет повторять вычисление произвольного выражения. Здесь e^3_i является выражением, задающим очередное значение переменной, управляющей циклом, при каждом повторении цикла. При этом выражение e^3_i каждый раз вычисляется заново; обычно оно прямо или косвенно зависит от текущего значения переменной, управляющей циклом.

Выполнение цикла в соответствии со спецификацией, содержащей и конструкцию BY/С_ШАГОМ e^1_i, и конструкцию TO/ДО e^2_i, может быть представлено одной из двух последовательностей операторов (в зависимости от знака значения e^1_i):

При $e^1_i \geq 0$	При $e^1_i < 0$
$u = e^0_i$	$u = e^0_i$
$u_1 = e^1_i$	$u_1 = e^1_i$
$u_2 = e^2_i$	$u_2 = e^2_i$
goto m2;	goto m2;
m1:$u=u+u_1$;	m1:$u=u+u_1$;
m2:if $u>u_2$ then goto me;	m2:if $u<u_2$ then goto me
s_1	
s_2	
…	
s_n	
goto m1;	

me:; < переход к следующей спецификации цикла >

Примером использования этой спецификации может служить программа, печатающая значения синусов углов 0,1, 0,2, 0,3, ,,., 1:

```
psin2: proc main;                    psin2: проц главная;
do x = 0.1 to l by 0.1;              цикл x = 0.1 до l с_шагом 0.1;
        put list(sin(x));                    писать в_виде(sin(x));
end x;                               конец x;
end psin2;                           конец psin2;
```

Если конструкция e_i, задающая изменение переменной u, имеет вид TO $e^2{}_i$, то она эквивалентна конструкции TO $e^2{}_i$ BY 1. (Т.е. если шаг явно не указан, то он считается равным единице).

Спецификация $e^0{}_i$ BY $e^1{}_i$ задает цикл, который может быть представлен последовательностью операторов:

$u = e^0{}_i$

$u_1 = e^1{}_i$

goto m2;

m1:$u=u+u_1$;

m2:

s_1

s_2

…

s_n

goto ml;

Никакого ограничения цикла данная спецификация не задает; следовательно, выход из цикла должен быть осуществлен при выполнении какого-либо из операторов s_1, s_2... Подобные спецификации должны быть единственными или последними в списке спецификаций оператора DO/ЦИКЛ. Примером использования такой спецификации может служить программа, в которой вводятся числа, и подсчитывается их количество:

```
pnum: proc main;                     pnum: проц главная;
dcl x float, i fixed(15);            опс x вещ, i точное(15);
on endfile(sysin) goto me;           когда конец_файла(стд_ввод) идти me;
do i = 0 by 1;                       цикл i = 0 с_шагом 1;
        get list(x);                         читать в_виде(x);
end i;                               конец i;
me: put list(i);                     me: писать в_виде(i);
end pnum;                            конец pnum;
```

Здесь выход из цикла обеспечивается оператором:

on endfile (sysin) goto me; или когда конец_файла(стд_ввод) идти me;

который задает передачу управления на метку me по достижении конца входных данных.

Спецификация REPEAT/ПОВТОРЯЯ (e^3_i) задает цикл, который может быть представлен последовательностью операторов:

$u = e^0_i$

m1:

s_1

s_2

…

s_n

$u = e^3_i$

goto m1;

И эта спецификация не задает никакого ограничения цикла, и, следовательно, выход должен быть осуществлен при выполнении какого-либо из операторов s_1, s_2, …, s_n. Примером использования такой спецификации может служить программа, в которой вводятся числа, и подсчитывается их сумма:

```
psm: proc main;                  psm: проц главная;
dcl (s,x) float(53);             опс (s,x) вещ(53);
on endfile(sysin) goto me;       когда конец_файла(стд_ввод) идти me;
do s = 0 repeat (s+x);           цикл s = 0 повторяя (s+x);
get list(x);                     читать в_виде(x);
end s;                           конец s;
me: put list(s);                 me: писать в_виде(s);
end psm;                         конец psm;
```

Здесь также выход из цикла обеспечивается оператором

on endfile (sysin) goto me; или когда конец_файла(стд_ввод) идти me;

по достижении конца входных данных. Если оттранслировать и запустить такую программу и с клавиатуры вводить числа, нажимая поле каждого Enter, а в конце всего ввода Ctrl+Z (имитация конца файла ввода), то на экран будет выведена их сумма.

Конструкция c_i, задающая условие выполнения цикла, имеет вид:

WHILE/ПОКА (e^4_i)

Смысл конструкций WHILE/ПОКА здесь тот же, что и в операторах начала циклической группы 1-го типа.

Формально допустима спецификация вида $e^0{}_i$ WHILE/ПОКА ($e^4{}_i$) , она задает цикл, который либо будет выполняться бесконечно (если $e^4{}_i$ истинно), либо вообще не будет выполнен. Порядок выполнения такого цикла может быть представлен следующей эквивалентной последовательностью операторов:

$u = e^0{}_i$

m1: if ^($e^4{}_i$) then goto me;

s_1

s_2

…

s_n

goto m1;

me:; < переход к следующей спецификации цикла >

на практике спецификации данного вида применяются редко.

В спецификациях цикла вида:

$e^0{}_i$ BY $e^1{}_i$ WHILE ($e^4{}_i$)

$e^0{}_i$ REPEAT($e^3{}_i$) WHILE ($e^4{}_i$)

перед каждым очередным повторением цикла изменяется значение переменной u. Условия выхода из цикла определяются конструкцией WHILE/ПОКА.

В качестве примера, иллюстрирующего применение рассмотренных спецификаций, приведем два варианта программы, вычисляющей сумму:

$$\sum_{n=1}^{N} n^x \ (x<0))$$

где N определяется из условий: $N^x > a$, $(N + 1)^x < a$.

Требуемая программа может иметь вид

Вариант 1

```
pnx: proc main;
dcl (a,x,s) float, n fixed;
get list(x, a);
s =0;
do n = 1 by 1 while(n**x > a);
    s = s + n**x;
end n;
put list(s);
end pnx;
```

```
pnx: проц главная;
опс (a,x,s) вещ, n точное;
читать в_виде(x, a);
s =0;
цикл n = 1 by 1 пока(n**x > a);
    s = s + n**x;
конец n;
писать в_виде(s);
конец pnx;
```

Вариант 2

```
pnx: proc main;
dcl (a,x,s,d) float, n fixed;
get list(x, a);
s = 0;
do d = 1 repeat(n**x) while (d > a);
    s = s + d;
    n=n+1;
end d;
put list(s);
end pnx;
```

```
pnx: проц главная;
опс (a,x,s,d) вещ, n точное;
читать в_виде(x, a);
s = 0;
цикл d = 1 повторяя(n**x) пока (d > a);
    s = s + d;
    n=n+1;
конец d;
писать в_виде(s);
конец pnx;
```

Наконец рассмотрим наиболее общую форму спецификации цикла:

$e^0{}_i$ BY/C_ШАГОМ $e^1{}_i$ TO/ДО $e^2{}_i$ WHILE/ПОКА ($e^4{}_i$)

В этой спецификации если $e^1{}_i$ равно 1, то конструкция BY/C_ШАГОМ $e^1{}_i$ может быть опущена.

Циклы, определяемые рассматриваемыми спецификациями, могут быть прерваны и тогда, когда значение переменной u выйдет за предел, заданный выражением $e^2{}_i$, и при невыполнении условий, указанных в конструкциях WHILE/ПОКА.

В качестве примера использования рассмотренных спецификаций цикла приведем программу, которая вычисляет сумму:

$$\sum_{n=1}^{N} \frac{1}{n}$$

в которой учтен тот факт, что если для какого-либо n представленное в памяти значение $\frac{1}{n}$ стало равным нулю, то продолжать процесс суммирования бессмысленно. Искомая программа может иметь вид

Вариант 1

```
csn: proc main;
dcl (s,y) float(53), (i,n) fixed(63);
get list(n);
s = 0;
y = l;
do i = 1 to n while(y > 0);
   s = s + y;
   y=1e0/(i+1);
end i;
put list(s);
```

```
csn: проц главная;
опс (s,y) вещ(53), (i,n) точное(63);
читать в_виде(n);
s = 0;
y = l;
цикл i = 1 до n пока(y > 0);
   s = s + y;
   y=1e0/(i+1);
конец i;
писать в_виде(s);
```

```
 end csn;                                  конец csn;
```

Вариант 2

```
 csn: proc main;                           csn: проц главная;
 dcl s float(53), (i, n) fixed(63);        опс s вещ(53), (i, n) точное(63);
 get list(n);                              читать в_виде(n);
 s = 0;                                    s = 0;
 do i = 1 to n while(1e0/i > 0);           цикл i = 1 до n пока(1e0/i > 0);
        s=s+1e0/i;                                s=s+1e0/i;
 end i;                                    конец i;
 put list(s);                              писать в_виде(s);
 end csn;                                  конец csn;
```

На практике часто бывает необходимость окончания цикла или необходимость начать новое выполнение цикла, не доходя до конца циклической группы. Особенно часто это бывает в сложных циклических группах, где подобные решения определяются в самой последовательности операторов s_1, s_2, ..., s_n. Учитывая это, в язык PL/1 ведены специальные оператор прекращения или повторения цикла. Они имеют вид

LEAVE/ХВАТИТ **CONTINUE/ОПЯТЬ**

При этом сами операторы должны входить в состав данной циклической группы. Рассмотренную выше программу при использовании оператора LEAVE/ХВАТИТ можно представить в виде

```
 csn: proc main;                           csn: проц главная;
 dcl (s, y) float(53), (i, n) fixed(63);   опс (s,y) вещ(53), (i, n) точное(63);
 get list(n);                              читать в_виде(n);
 s = 0;                                    s = 0;
 do i = 1 to n;                            цикл i = 1 до n;
    y=1e0/i;                                      y=1e0/i;
    if y=0 then leave;                            если y=0 тогда хватит;
     s=s+y;                                       s=s+y;
 end i;                                    конец i;
 put list(s);                              писать в_виде(s);
 end csn;                                  конец csn;
```

Введение в циклические группы в языке PL/1 операторов LEAVE/ХВАТИТ и CONTINUE/ОПЯТЬ связано в основном с тем, что, как показала практика, использование обычных операторов перехода в циклах уменьшает надежность программирования.

В заключение отметим следующее ограничение языка PL/1.

С помощью оператора перехода, находящегося вне некоторой циклической группы, нельзя передать управление вовнутрь этой группы независимо от того, был начат цикл выполнения этой группы или нет.

Проверь себя

1. Всегда ли группа операторов, в начале которой стоит оператор DO/ЦИКЛ, является циклической?
2. Чему будет равно значение переменной N после выполнения каждой из приведенных ниже последовательностей операторов, при условии, что вначале значение N равно нулю?

```
1) do while(n < 10);
       n = n + 3;
   end;
```

```
2) цикл пока(n < 0);
       n = n + 3;
   конец;
```

```
3) do i=3, 5, 8;
       n=n+i;
   end;
```

```
4) цикл i =2 до 9 с_шагом 2;
       n=n+1;
   конец;
```

```
5) do i = 2 to 9;
       n +=1;
   end;
```

```
6) цикл i = 0 с_шагом 2;
   если  i > n+10 тогда хватит;
       n +=1;
   конец;
```

```
7) do i =2 while(n < 10);
       n +=1;
   end;
```

```
8) цикл i =2 до 6 пока(n < 10);
   n +=1;
   конец;
```

```
9) цикл n= 1 повторяя(2*n) пока(n < 100);
   end;
```

3. Сколько раз будет выполнен следующий цикл:

```
цикл i = 1 с_шагом(i + 1) до(10*i —i**2); end;
```

4. Для чего предназначен оператор LEAVE/ХВАТИТ?
5. Для чего предназначен оператор CONTINUE/ОПЯТЬ?
6. В чем состоит ошибка в приведенной ниже программе?

```
a: proc main;
…
do while(x^=5);
…
    goto m1;
м2:x=x + 3;
end while;
m1: x -=2;
goto м2;
```

```
a: проц главная;
…
цикл пока(x^=5);
…
    идти m1;
м2:x=x + 3;
конец пока;
m1: x -=2;
идти м2;
```

end a; конец a;

7. Используя циклические группы, составить программы для вычисления следующих величин:

$$x = \sum_{i=0}^{100} \sin(i + 3.14)$$

$$y = \prod_{i=1}^{50} \lg(i + 0.5)$$

$z = \sum_{n=1} 1/(nx)^2$ для всех n, удовлетворяющих условию $1/(nx)^2 > a > 0$.

3.2. Массив. Описание массива

В языке PL/1 переменные можно объединять в совокупности, называемые массивами. Значения всех переменных, входящих в один массив, должны иметь одинаковые характеристики. Например, если массив состоит из числовых переменных, то все переменные должны быть либо только комплексного, либо только вещественного типа, с одинаковой формой представления, основанием и точностью значений. Переменные, входящие в некоторый массив, являются элементами данного массива.

Каждый массив имеет некоторое имя. Именем массива, не входящего в какую-либо объемлющую совокупность, может быть любой идентификатор, выбираемый программистом (пользователем) по своему усмотрению.

Переменную, входящую в состав массива, называют *переменной с индексами* (либо индексированной переменной). Каждой переменной u с индексами, входящей в некоторый массив A, взаимно однозначно сопоставляется n целых чисел $i_1, i_2, \ldots, i_n$, называемых *индексами* переменной u относительно массива A. Число n одинаково для всех переменных, входящих в массив A, оно называется *размерностью массива* A. Число i_j $(1 \le j \le n)$ называется индексом переменной u по j-измерению массива A.

Для любой переменной, входящей в массив A, значение индекса по j-измерению не меньше некоторого фиксированного числа T_j^1 и не больше некоторого фиксированного числа T_j^2. Число T_j^1 называется нижней границей индекса j-го измерения массива A, числоT_j^2 — верхней границей индекса j-го измерения.

Для любой последовательности целых чисел $i_1, i_2, \ldots, i_n$ удовлетворяющих условию $T_1^1 \le i_1 \le T_1^2$, $T_2^1 \le i_2 \le T_2^2$ ..., $T_n^1 \le i_n \le , T_n^2$, где T_j^1 и T_j^2 $(j=1, 2, \ldots, n)$ — нижняя и верхняя граница j-го измерения n-мерного массива A, имеется переменная с индексами $i_1, i_2, \ldots, i_n$, входящая в массив A.

Таким образом, массив *A* состоит из $(T_1^2 - T_1^1 + 1)*(T_2^2 - T_2^1 + 1)* \ldots *(T_n^2 - T_n^1 + 1)$ переменных.

Приведем пример. Пусть имеется одномерный массив *A* с границами индекса 1 и 4, а также двумерный массив *B* с границами индекса 0 и 2 по первому измерению, -1 и 1 — по 2-му измерению. Массив *A* состоит из 4 переменных, каждой из которых сопоставлен, соответственно, индекс 1, 2, 3 и 4. Массив *B* состоит из 3x3=9 переменных, каждой из которых сопоставлена пара индексов (0, -1), (0, 1), (0, 1), (1, -1), (1, 0), (1, 1), (2, -1), (2, 0), (2, 1).

В рассматриваемой версии языка PL/1 имеются следующие ограничения:

- размерность массива должна удовлетворять соотношению $1<n<15$;
- границы индекса T^1 и T^2 должны удовлетворять соотношению

$-2147483648 \le T^1 \le T^2 \le 2147483647$.

Переменная с индексами (не входящая в состав структуры — см. гл. 7) представляется в PL/1-программе в виде

$$a\ (e_1, e_2, \ldots, e_n)$$

где a — имя массива, содержащего данную переменную; n — размерность этого массива; e_i $(i = 1, 2,.., n)$ — выражение, называемое индексным. Значение каждого индексного выражения должно быть целым числом, иначе оно автоматически преобразуется в целое число по стандартным правилам языка PL/1.

Пусть значения выражений $e_1, e_2, \ldots, e_n$ равны целым числам $i_1, i_2, \ldots, i_n$ и пусть выполняются соотношения $T_1^1 \le i_1 \le T_1^2$, $T_2^1 \le i_2 \le T_2^2$ ..., $T_n^1 \le i_n \le , T_n^2$, где T_j^1 и T_j^2 $(j=1, 2, ..., n)$ — нижняя и верхняя граница индекса j-го измерения массива с именем a. Тогда конструкция $a\ (e_1, e_2, \ldots, e_n)$ в PL/1-программе представляет переменную, входящую в массив с именем a и имеющую индексы $i_1, i_2, \ldots, i_n$. Например, пусть переменные x и k имеют значения, соответственно, 1,9 и 2, пусть двумерный массив *A* имеет границы 1 и 3 по каждому измерению. Тогда любая из конструкций

A (2, 2); A (2*K—2, +2); A(X+1, K);

представляет собой обращение к переменной, входящей в массив *A* и имеющей индексы 2 и 2, и называется именем переменной с индексом. Конструкция A(X-1, K) не является допустимым именем переменной с индексом, так как окончательное значение первого индексного выражения равно нулю и меньше нижней границы индекса 1-го измерения массива *A*.

Переменные, являющиеся элементами массивов, могут быть использованы почти во всех конструкциях языка PL/1, в которых допускается использование простых переменных. В частности, они могут входить в состав

выражений, в левую часть операторов присваивания, в список данных оператора GET/ЧИТАТЬ. Элементами массивов могут быть также переменные, управляющие циклом. В последнем случае индексные выражения, входящие в имя переменной, управляющей циклом, вычисляются заранее до начала цикла и в процессе выполнения цикла не пересчитываются.

Переменные, объединенные в массив, обычно используются в тех случаях, когда одни и те же действия должны быть выполнены над значениями нескольких однотипных переменных. Рассмотрим в качестве примера задачу вычисления суммы

$\sum_{i=1}^{N/2} a_i \ a_{N-i+1}$, где числа $a_1, a_2, \ldots, a_n$ вводятся.

Если нельзя переставлять вводимые данные, то задача может быть решена только при условии хранения в памяти всех или, по крайней мере, половины вводимых значений. Если бы N было фиксировано и невелико, то задачу можно было бы решить и без использования массивов. Однако при больших N решение этой задачи без использования массивов становится практически трудной. Предположим, что N заведомо не превышает 100 и пусть A — одномерный массив с границами изменения индекса от 1 да 100. Тогда требуемая сумма будет вычислена с помощью операторов

```
on endfile(sysin) goto м;
dcl (n,i) fixed, a(100) float;
do n= 1 by 1;
    get list a(n);
end n;
m: sum = 0;
n = n-1;
do i=1 to n/2;
     sum+=a(i)*a(n-i+1);
end;
```

```
когда конец_файла(стд_ввод) идти м;
опс (n,i) точное, a(100) вещ;
цикл n= 1 с_шагом 1;
    читать в_виде a(n);
конец n;
m: sum = 0;
n = n-1;
цикл i=1 до n/2;
     sum+=a(i)*a(n-i+1);
конец;
```

Отметим также, что при решении математических задач одномерные массивы обычно используются для представления векторов, а двумерные массивы — для представления матриц.

Каждый массив, используемый в программе, должен быть описан в операторе описания. Описание массива имеет вид:

$$tda_1a_2\ldots a_m \quad (m \geq 0)$$

Здесь t — идентификатор, являющийся именем массива; d — описатель измерений; $a_1, a_2, \ldots, a_m$ — прочие описатели, в частности, описатели типа переменных, образующих массив. Описатель измерений имеет вид

$$(b_1, b_2, \ldots, b_n)$$

где конструкция b_k $(k = 1, 2, ..., n)$, описывающая k-е измерение массива, может иметь одну из форм:

1) T_k^2
2) $T_k^1 : T_k^2$
3) *

Здесь T_k^1 — выражение, значение которого определяет нижнюю границу индекса k-го измерения; T_k^2 — выражение, определяющее его верхнюю границу (использование звездочки будет рассмотрено в § 9.1 и § 13.2). Если T_k^1 не задано, то считается, что нижняя граница равна 1.

Тип переменных, образующих массив, задается с помощью описателей типа по тем же правилам, что и для простых переменных. В частности, если ни одного описателя типа не задано, то по умолчанию считается, что переменным приписаны описатели fixed binary (15) или точное двоичное(15).

Рассмотрим примеры описаний массивов.

1. Пусть массив с именем A состоит из 5 переменных типа decimal float (6), имеющих индексы 1, 2, 3, 4, 5. Тогда массив может быть описан с помощью оператора описания:

dcl a(1 : 5) decimal float (6); или опс a(1:5) десятичное вещ(6);

2. Пусть массив с именем num состоит из 6 переменных типа fixed binary (15, 0), имеющих пары индексов: (1, 1), (1,2), (1,3), (2, 1), (2,2), (2, 3). Массив может быть описан с помощью следующего оператора описания:

dcl num(1:2, 1:3) fixed binary (15); или опс num(1:2, 1:3) точное(15);

С учетом правил умолчания данному оператору будут эквивалентны следующие операторы описания:

dcl num (2, 3) fixed(15); опс num (2, 3) точное(15);
dcl num (2, 3); опс num (2, 3);

3. Пусть массив с именем DR состоит из 8 переменных типа complex float, имеющих по 3 индекса: (0, -1, -2), (0,-1, -1), (0, 0, -2), (0, 0, -1), (1,-1, -2), (1,-1, -1), (1, 0, -2) и (1, 0, -1). Массив может быть задан с помощью оператора описания вида

dcl dr(0:1 ,-1:0, -2:-1) complex float; опс dr(0:1 ,-1:0, -2:-1) комплексное вещ;

Если в одном операторе описания описывается несколько массивов, имеющих одинаковые описатели, или массивы и простые переменные с одинаковыми описателями, то такие общие описатели могут быть вынесены за скобки аналогично тому, как это делалось в § 2.2 для простых переменных. Например, оператор:

dcl a(5) complex float, b(5) complex float, x complex float;
опс a(5) комплексное вещ, b(5) комплексное вещ, x комплексное вещ;

может быть записан в краткой форме:

```
dcl (a(5), b(5), x) complex float;
опс (a(5), b(5), x) комплексное вещ;
```

Наряду с другими описателями можно выносить за скобки и общие описатели измерений массивов. Например, рассмотренный выше оператор может быть записан в форме

```
dcl ((a, b) (5), x) complex float;  опс ((a, b) (5), x) комплексное вещ;
```

Проверь себя

1. Можно ли объединить в один массив переменные с комплексными и действительными значениями?
2. Что называется размерностью массива?
3. Может ли одномерный массив содержать две переменных с одинаковым индексом?
4. Пусть нижняя граница индекса обоих измерений двумерного массива равна 0, а верхняя граница равна 2. Сколько переменных входит в массив? Перечислить последовательности индексов всех элементов массива.
5. Какие ограничения на значение индекса имеются в рассматриваемой версии языка PL/1?
6. Пусть границы индекса одномерного массива *A* равны 1 и 5. Какие из следующих имен являются правильными именами элементов массива *A*, при условии, что значение переменной *N* равно 2?

```
a(n);   a(n + 5);   a(n*2);   a(n/2);   a(n-l.l);   a(n + 3.9).
```

7. Каково будет значение переменных A(1) и A(2) после выполнения операторов

```
n, a(1), a(2) = 2;
n, a(l), a(a(n)) = 1;
```

8. Каково будет значение переменных A(1), A(2), A(3), A(4) после выполнения операторов

```
n = 1;
do a(n) = 1 то 5 by 2;
n,a(n+1) = 1;
end;
```

9. Составьте операторы описания для следующих массивов числовых переменных:

1) одномерный массив N с границами индекса от 1 до 10, содержащий переменные с фиксированной точкой и точностью (15, 0);

2) двумерный массив R с границами индекса от 0 до 5 по обоим измерениям, содержащий переменные с комплексными значениями с плавающей точкой и точностью (53);

3) двумерный массив K с границами индекса от 1 до 5 и от 0 до 100, содержащий переменные с десятичными значениями с фиксированной точкой и точностью (15, 0);

4) трехмерный массив S с границами индекса от -1 до 1 по всем измерениям, содержащий переменные с значениями с плавающей точкой и точностью (24).

10. Перепишите в наиболее краткой форме следующие операторы описания

1) dcl a(1:4), b(4), c(4) float(24);

2) dcl p(5) decimal(15,0), r(5) binary(15);

3.3. Действия с массивами

Характерной особенностью языка PL/1 является возможность задавать действия не только над элементами массивов, но иногда и над целыми массивами. В частности, массивы можно указывать в списках данных операторов GET/ЧИТАТЬ и PUT/ПИСАТЬ, в операторах присваивания, в списках аргументов функций. Например, ввод всех значений элементов массива X можно задать с помощью оператора:

get list(x); или читать в_виде(x);

Если *x* — одномерный массив с границами индекса от 1 до 100, то выполнение указанного оператора будет иметь тот же эффект, что и выполнение цикла:

do i = 1 то 100; get list(x(i)); end; цикл i = 1 до 100; читать в_виде(x(i)); конец;

Для того чтобы правильно подготовить исходные данные, нужно учитывать, в каком порядке значения будут присваиваться элементам массива. Поэтому в данном случае, как и в ряде других случаев работы с массивами, необходимо знать следующее общее правило.

Выполнение любого действия над целым массивом всегда сводится к последовательному выполнению действий над элементами массивов, взятых в следующем порядке: вначале берется элемент с минимальными значениями всех индексов, затем последовательно увеличивается значение крайнего справа индекса, затем увеличивается на 1 значение предпоследнего индекса, а крайний справа индекс вновь меняется от нижней до верхней границы и т.д.; последним берется элемент с максимальными значениями всех индексов.

Например, для одномерного массива с границами от l до $l+k$ элементы берутся в следующем порядке: a_l, a_{l+1}, …, a_{l+k}; для двумерного массива с границами от 1 до m и от 1 до n элементы берутся в следующем порядке:

$$a_{1,1}, a_{1,2}, \ldots, a_{1,n}, a_{2,1}, a_{2,2}, \ldots, a_{2,n}, \ldots, a_{m,1}, a_{m,2}, \ldots, a_{m,n}$$

Для выполнения одинаковых операций над всеми элементами одного или нескольких массивов удобно использовать выражения над массивами.

В рассматриваемой версии языка PL/1 нельзя выполнять произвольные действия над массивами, но можно:

1) Присваивать массив массиву. В этом случае размерности и характеристики элементов массива справа и слева в присваивании должны совпадать. Кроме присваивания, возможно использование и оператора обмена, меняющего содержимое массивов местами. Например, пусть A и B массивы из миллиона элементов с плавающей точкой, тогда оператор a <=> b; поменяет значение каждого из этого миллиона элементов на соответствующее из другого массива и наоборот. При этом никаких «промежуточных» массивов создаваться не будет.

2) Присваивать массив нулю одним оператором. Все элементы массива получат нулевое значение.

3) Сравнение массивов на равенство (=) или неравенство (^=) одним оператором. Массивы считаются равными, если все элементы одного побайтно совпадают с элементами другого и не равными в остальных случаях.

4) Сравнение массива с нулем одним оператором. Массив считается равным нулю, если все его элементы содержат нулевые значения и не равным нулю в остальных случаях.

Проверь себя

1. Какие действия над массивами можно выполнять одним оператором в рассматриваемой версии языка?
2. Допустимо ли множественное присваивание массивов? Пусть A, B, C — массивы одинаковой размерности и характеристик элементов. Допустим ли оператор A,B=C;?
3. При выводе массива оператором PUT DATA/ПИСАТЬ С_ИМЕНАМИ будут ли автоматически выводиться значения индексов для каждого выводимого элемента массива?
4. Можно ли сравнить на равенство одним оператором два массива одинаковой размерности, если элементы одного имеют точность вещ(24), а второго — вещ(53)?

3.4. Организация циклов внутри операторов ввода и вывода

В целях облегчения составления программы предусмотрена возможность задания циклов непосредственно внутри списков данных операторов ввода (GET/ЧИТАТЬ) и вывода (PUT/ПИСАТЬ).

Рассмотрим следующий пример. Пусть имеется двумерный числовой массив *A* с диапазоном индексов по обоим измерениям от 1 до 20. Требуется вывести значения элементов «главной диагонали» массива, т.е. элементов a(i,j), где i=j. Это можно сделать посредством операторов:

```
do i = 1 to 20;                    цикл i = 1 до 20;
   put list(a(i, i));                  писать в_виде(a(i, i));
end i;                             конец i;
```

Однако эта задача решается и одним оператором вывода:

```
put list((a(i, i) do i = 1 to 20));        писать в_виде((a(i, i) цикл i = 1 до 20));
```

Заметим, что вместо констант нижних и верхних границ индексов циклов в подобных операторах удобнее использовать встроенные функции HBOUND/ВЕРХ_ГРАНИЦА, LBOUND/НИЖ_ГРАНИЦА, DIMENSION/РАЗМЕРНОСТЬ, которые рассматриваются далее.

Список данных операторов GET/ЧИТАТЬ или PUT/ПИСАТЬ может в общем случае иметь вид:

$$(a_1, a_2, \ldots, a_n)\ (n \geq 1)$$

Здесь $a_i\ (i = 1, 2, \ldots, n)$ — либо одиночный элемент списка данных, либо повторяемый подсписок данных, имеющий вид

$$(a'_1, a'_2, \ldots, a'_m\ DO\ u = s_1, s_2, \ldots, s_k)\ \ (m \geq 1, k \geq l)$$

где $a_j\ (j=1, 2, \ldots, m)$ — также либо одиночный элемент списка данных, либо повторяемый подсписок данных; u — переменная, управляющая циклическим повторением подсписка; $s_l\ (l= 1, 2, \ldots, k)$ — спецификация цикла, полностью аналогичная допустимым спецификациям соответствующей формы оператора DO/ЦИКЛ (см. § 3.1). Если в процессе ввода или вывода в соответствии с некоторым списком данных в нем встречается повторяемый подсписок данных, то далее ввод или вывод будут производиться в соответствии с этим подсписком, просматриваемым целиком столько раз, сколько задают указанные в нем спецификации цикла. По окончании ввода или вывода в соответствии с повторяемым подсписком производится переход к следующему элементу списка данных.

Например, при выполнении оператора:

```
писать в_виде(((a(i, j) цикл i = 1 до m), с_новой цикл j = 1 до n));
```

где *A* — двумерный массив с границами индекса — *(1:m, 1:n)*, будут выводиться значения элементов каждого столбца массива, а за ними —

перевод строки, и так для каждого столбца массива. Обратите внимание на то, что каждый повторяемый подсписок должен быть заключен в свои собственные круглые скобки. Именно поэтому в приведенном выше примере в начале и конце списка данных стоит по паре скобок.

Повторяемые подсписки допустимы во всех рассмотренных ранее вариантах операторов ввода и вывода, т. е. в операторах GET/ЧИТАТЬ и PUT/ПИСАТЬ с ключевым словом LIST/В_ВИДЕ и в операторах PUT/ПИСАТЬ с ключевым словом DATA/С_ИМЕНАМИ.

Проверь себя

1. Каков практический смысл задания циклов внутри операторов ввода или вывода?
2. Какие данные и в какой последовательности будут выведены следующим оператором при условии, что C — двумерный числовой массив с диапазоном индекса по обоим измерениям от 1 до 40:

```
писать в_виде(((a(i, j) цикл j = 40 с_шагом -1 до 1), с_новой цикл i =40 с_шагом - 1 до 1));
```

3. Какова синтаксическая ошибка в следующем операторе (j — имя числовой переменной):

```
put list (exp (j) do j = 1 to 30);
```

4. ПРОЦЕДУРЫ, ФУНКЦИИ, БЛОКИ

4.1. Связь между процедурами

Программа на языке PL/1 может состоять из нескольких программных единиц, называемых процедурами. Деление программы более чем на одну процедуру полезно, в основном, в следующих двух случаях. Во-первых, когда программа является настолько большой или сложной, что ее становится рациональнее составлять и отлаживать по частям. Во-вторых, если одни и те же достаточно сложные действия должны быть выполнены в различных точках программы, в том числе и в случае, когда в разные моменты эти действия должны выполняться над различными объектами — переменными, массивами и т.д.

Текст какой-либо процедуры может быть целиком включен в текст другой процедуры. В этом случае первая процедура называется *внутренней* по отношению к объемлющей процедуре. Процедуру, не являющуюся внутренней по отношению ко всем другим процедурам, назовем *внешней.* В общем случае программа может состоять из нескольких внешних процедур, но только одна из них считается главной; именно с этой процедуры начинается выполнение программы.

Любая процедура начинается с оператора заголовка процедуры, имеющего в общем случае вид

PROCEDURE $pa_1a_2, ..., .a_n$ $(n \geq 0)$ **ПРОЦЕДУРА** $pa_1a_2, ..., .a_n$ $(n \geq 0)$

где p — либо пусто, либо список параметров процедуры (см. ниже); a_i $(i = 1, 2, ..., n)$ — дополнительные конструкции, описывающие различные свойства процедуры. В их число может входить конструкция **MAIN/ГЛАВНАЯ**, задаваемая для главных процедур.

В данной главе будут также рассмотрены такие дополнительные конструкции этого оператора, как **RECURSIVE/РЕКУРСИВНАЯ** и **RETURNS/ВОЗВРАЩАЕТ**. Порядок конструкций a,- может быть произвольным. Для ключевого слова PROCEDURE/ПРОЦЕДУРА допустимо сокращение **PROC/ПРОЦ**.

Перед любым оператором заголовка процедуры должна стоять одна метка. Метка, стоящая перед такими операторами, называется *именем входа* в данную процедуру. В конце каждой процедуры должен стоять оператор конца (END/КОНЕЦ) после него, но перед точкой с запятой, можно повторить имя процедуры для контроля правильности расстановки операторов конца.

Если в процессе выполнения объемлющей процедуры будет выполнен оператор, предшествующий заголовку внутренней процедуры, и этот оператор

не изменит порядка выполнения, то следующим будет выполнен оператор, стоящий за оператором конца внутренней процедуры. Поэтому текст внутренней процедуры может быть расположен в любом месте объемлющей процедуры.

При выполнении какой-либо процедуры можно передать управление другой процедуре. В этом случае первая процедура называется *вызывающей,* а вторая — *вызываемой.* Процесс передачи управления в данном случае называется *вызовом процедуры.* Процедура может быть вызвана либо посредством указателя функции, либо путем выполнения специального оператора вызова процедур, имеющего в простейшем случае вид

CALL *fa* или **ВЫЗОВ *fa*** или просто ***fa***

где *f* — имя входа в вызываемую процедуру; *a* — либо пусто, либо список аргументов (см. ниже).

Передавать управление оператору заголовка процедуры с помощью оператора перехода нельзя.

Процедуру, выполнение которой было начато, но не закончено, принято называть *активной.* В каждый момент времени может иметься несколько активных процедур, в частности, при вызове вызываемая процедура становится активной и вызывающая процедура остается активной. Имеется следующее требование: может быть вызвана лишь процедура, либо являющаяся внешней, либо непосредственно входящая в активную процедуру. Более того, вообще никаким способом нельзя передать управление внутрь процедуры, не являющейся активной.

Если в программе требуется вызывать процедуру, которая к моменту вызова уже является активной, то такая процедура называется *рекурсивной* и в число дополнительных конструкций оператора ее заголовка должно входить ключевое слово RECURSIVE/РЕКУРСИВНАЯ.

Возврат управления из вызванной процедуры в вызвавшую осуществляется при достижении оператора конца вызванной процедуры. Кроме того возврат управления может быть произведен с помощью оператора возврата, имеющего в простейшем случае вид RETURN/ВОЗВРАТ

При возврате управления вызванная процедура перестает быть активной. Выполнение процедуры, в которую было возвращено управление одним из рассмотренных способов, продолжается с оператора, следующего за оператором вызова.

Еще одним способом завершения процедуры является передача управления с помощью оператора перехода в любое место любой активной процедуры, а не только той, которая вызвала данную процедуру. При этом перестают быть активными все процедуры, вызванные из той, в которую

возвращено управление. Выполнение программы в этом случае продолжается, начиная с оператора, получившего управление.

Наконец, в любой процедуре можно завершить выполнение всей программы посредством *оператора останова,* состоящего только из ключевого слова **STOP/СТОП**. Есть форма этого оператора, с целочисленным значением в круглых скобках. В этом случае, указанное число передается как код ответа операционной системе.

Передача информации между процедурами обычно осуществляется с помощью аппарата параметров. *Формальный параметр* (или просто параметр) процедуры представляет собой идентификатор, который в тексте данной процедуры используется в качестве условного обозначения произвольной переменной определенного типа, произвольного массива заданной размерности, содержащего переменные определенного типа, а также для обозначения других объектов, допустимых в программах на языке PL/1 и рассмотренных в последующих главах. Этот идентификатор не обязан совпадать с именем какой-либо конкретной переменной, конкретного массива и т.д. Более того, начинающим программистам можно рекомендовать использовать в качестве формальных параметров идентификаторы, отличные от любых имен, имеющихся в вызывающей процедуре.

Оператор заголовка процедур, имеющих параметры, включает список параметров: $(p_1, p_2, \ldots, p_n), \ (n \geq 1)$

где $p_i \ (i=1, 2, \ldots, n)$ — идентификатор, являющийся именем формального параметра. Параметр обязательно должен быть описан в операторе описания непосредственно в тексте вызываемой процедуры. В описателе измерений массивов-параметров в качестве выражений для границ допускаются только константы. Можно в некоторых случаях для границ указывать звездочки (или задать звездочки для старшей части измерений), число которых должно быть не больше размерности массива.

При выполнении процедуры во всех случаях, когда указан формальный параметр, будет использоваться конкретная переменная (массив, входа в процедуру и т.д.), задаваемая при вызове процедуры и называемая *аргументом* процедуры. Аргументы перечисляются в списке (см. выше описание оператора CALL/ВЫЗОВ), имеющем вид

$$(a_1, a_2, \ldots, a_n) \quad (n \geq 0)$$

Здесь $a_i \ (i=1, 2, \ldots, n)$ является аргументом, который сопоставляется i-му формальному параметру процедуры. Число аргументов должно совпадать с числом параметров, указанных в соответствующем входе в процедуру.

Если аргумент, соответствующий параметру-переменной, является именем переменной, то обычно можно считать, что любое действие над параметром (в частности, присвоение ему значений) на самом деле предполагает точно такие же действия над аргументом. То же самое относится и к аргументам-массивам, сопоставляемым параметрам-массивам. Если же аргумент, сопоставляемый параметру-переменной является выражением, отличным от имени переменной, то при трансляции программы автоматически создается вспомогательная переменная (ее имя недоступно программисту), которой присваивается значение аргумента и которая фактически и сопоставляется параметру. Таким образом, в вызываемой процедуре имя любого формального параметра-переменной может быть использовано в левой части операторов присваивания. Подобные вспомогательные переменные называются фиктивными аргументами.

Допустимый вид аргумента, сопоставляемого некоторому параметру, зависит от типа этого параметра.

Главная процедура может иметь только один параметр типа строка, аргумент подставляет в который операционная система.

Главная процедура может иметь оператор возврата, который возвращает управление операционной системе (как и оператор СТОП(n)). При таком возврате главная программа может передать операционной системе и числовой значение — код ответа.

Вообще говоря, должны выполняться следующие требования:

– если параметр является переменной, то аргумент должен быть скалярным выражением со значением точно того же типа, что и тип переменной-параметра;

– если параметр является массивом, то аргумент должен быть массивом со значением той же размерности, состоящим из переменных точно того же типа, что и у массива-параметра.

Однако допускаются случаи, когда аргумент (не массив) не отвечает указанным требованиям; при этом автоматически создается вспомогательный, фиктивный аргумент, точно соответствующий типу параметра. Этому фиктивному аргументу присваивается значение фактического аргумента, далее вызванная процедура использует только фиктивный аргумент. Подчеркнем, что по окончании работы вызванной процедуры обратного присваивания значений фиктивного аргумента фактическому не производится. Поэтому, например, если фактический аргумент — имя переменной, то в случае создания фиктивного аргумента никакое изменение параметра не приводит к изменению исходного значения фактического аргумента.

В языке PL/1 при вызове внутренних процедур при несоответствии типа аргументов и параметров также всегда автоматически создаются фиктивные аргументы. При условии, конечно, что такой аргумент может в принципе быть создан, например, если аргумент — скалярное выражение, а параметр — массив, создание фиктивного аргумента становится невозможным. При вызове в языке PL/1 внешних процедур создание фиктивных аргументов из-за несоответствия типов аргументов и параметров выполняется на основании описателя **ENTRY/ДЛЯ_ВЫЗОВА** в вызывающей процедуре, задающего характеристики соответствующих параметров. Этот описатель задается в операторе DECLARE/ОПИСАНИЕ при описании имени входа, что будет рассмотрено в § 4.3. Наиболее частым случаем нарушения такого соответствия является использование в качестве аргументов констант, не соответствующих типу параметров. Например, для обработки целых чисел обычно применяются переменные (в том числе и параметры) типа FIXED BINARY. Изображаются же целые числа обычно с помощью десятичных констант.

В описываемой версии языка, возможно задание параметров-массивов с заранее неизвестными границами, в этом случае вместо значений границ указываются звездочки. Однако работа с такими массивами требует дополнительной подготовки, о которой будет рассказано в § 13.2.

В любом случае целесообразно и полезно стараться в операторах программы (например в границах заголовка цикла) явно не указывать границы массивов, даже, если они константы, а использовать функции **LBOUND (*x, k*), HBOUND (*x, k*) и DIM (*x, k*),** где x — массив, k — скалярное выражение с положительным целочисленным значением (значения других типов автоматически преобразуются к целому числу); значение k не должно превышать числа измерений массива x. Значение функции LBOUND *(x, k)* равно нижней границе k-го измерения массива x, значение функции HBOUND *(x, k)* равно верхней границе k-го измерения, значение функции DIM*(x, k)* равно увеличенной на 1 разности между верхней и нижней границами k-го измерения. Значения функций LBOUND, HBOUND и DIM являются вещественными двоичными числами с фиксированной точкой и точностью (31). Если k не задано, оно считается равным единице.

Русские эквиваленты названий этих функций – **НИЖ_ГРАНИЦА, ВЕРХ_ГРАНИЦА** и **РАЗМЕРНОСТЬ** соответственно.

Например, при вычислении суммы элементов массива A из 100 элементов вместо цикла

```
do i=1 to 100;  s +=a(i); end i;        цикл i=1 до 100;  s +=a(i); конец ;i
```

рациональнее написать

```
do i=lbound(a) to hbound(a);
    s +=a(i);
end i;
```

```
цикл i=ниж_граница(a) до верх_граница(a);
    s +=a(i);
конец i;
```

Что сделает данный фрагмент программы независимым при возможных изменениях значений границ-констант массива A/

Проверь себя

1. В каких основных случаях полезно делить программу на несколько процедур?
2. Какие процедуры называются внутренними, внешними, главными, вызывающими, вызываемыми, активными, рекурсивными?
3. Какой оператор будет выполнен вслед за оператором x=5; при выполнении следующей программы?

```
a: proc main;
x = 5;
b: proc;
y = 6;
…
end b;
z=7;
end a;
```

```
a: проц главная;
x = 5;
b: прос;
y = 6;
…
конец b;
z=7;
конец a;
```

4. Как может осуществляться вызов процедуры, возврат управления из процедуры?
5. Какие из следующих операторов перехода допустимы?

```
a: proc main;
…
goto м2;
…
ml: x = 5;
…
goto мз;
…
м2: proc;
…
мз: x = 6;
goto ml;
end m2;
end a;
```

```
a: проц главная;
…
идти м2;
…
ml: x = 5;
…
идти мз;
…
м2: прос;
…
мз: x = 6;
идти ml;
конец m2;
конец a;
```

6. В каких случаях тип аргумента может не соответствовать типу параметра?.

7. Что называется фиктивным аргументом? В каких случаях он создается?

8. Составьте процедуру с двумя параметрами, которая присваивает 1-у параметру значение 2-го параметра, а 2-у параметру — значение 1-го параметра. Используя эту процедуру, составьте программу, которая вводит три числа и печатает их в порядке возрастания.

4.2. Процедуры-функции

Ранее были рассмотрены встроенные функции языка PL/1, обращение к которым осуществляется с помощью указателей функций, являющихся операндами выражений. Программист может составить собственные процедуры, называемые процедурами-функциями, которые также будут вызываться с помощью указателей функций. Указатель функции, как и в случае встроенных функций, будет при этом иметь вид

fa

где f — имя точки входа в процедуру; a — либо пусто, либо список аргументов (такой же, как и в операторе CALL/ВЫЗОВ — см. § 4.1).

В процессе выполнения процедуры-функции должно быть вычислено некоторое значение, которое будет использовано в вызывающей процедуре при вычислении выражения на месте соответствующего указателя функции. Передача вычисленного значения в вызывающую процедуру осуществляется одновременно с возвратом управления с помощью оператора возврата, имеющего вид:

RETURN(*e*) или **ВОЗВРАТ(*e*)**

где e — скалярное выражение.

Если процедура была вызвана посредством указателя функции, то нельзя завершить ее работу ни с помощью оператора RETURN/ВОЗВРАТ без последующего выражения, ни с помощью оператора конца данной процедуры (однако допускается заканчивать выполнение процедуры с помощью оператора перехода, передающего управление в любую другую активную процедуру). С другой стороны, если процедура была вызвана посредством оператора CALL/ВЫЗОВ, то нельзя завершать ее работу с помощью оператора вида RETURN (*e*)/ВОЗВРАТ (*e*).

Для примера рассмотрим задачу вычисления периметра треугольника по координатам его вершины. Эта задача может быть решена, например, с помощью программы, состоящей из двух внешних процедур:

```
tr: proc main;
dcl ((x1, y1, x2, y2, x3, y3); float
get list(x1, y1, x2, y2, x3, y3);
put list(d(x1, yl, x2, y2)
        + d(x2, y2, x3, y3)
        + d(x3, y3, x1, y1));
end;
d: proc(xa, ya, xb, yb) returns(float);
return
(sqrt( (xa- xb)**2 + (ya-yb)**2));
end;
```

```
tr: проц главная;
опс (x1, y1, x2, y2, x3, y3) вещ;
читать в_виде(x1, y1, x2, y2, x3, y3);
писать в_виде(d(x1, yl, x2, y2)
                + d(x2, y2, x3, y3)
                + d(x3, y3, x1, y1));
конец;
d: проц(xa, ya, xb, yb) возвращает(вещ);
возврат
(sqrt( (xa- xb)**2 + (ya-yb)**2));
конец;
```

В данном случае использована процедура-функция D, значение которой равно расстоянию между двумя точками на плоскости.

Каждое значение, вычисляемое процедурой-функцией, должно относиться к определенному типу. Тип значения процедуры-функции должен быть явно определен и в вызываемой процедуре, и в вызывающей.

В операторе заголовка данной процедуры должна быть задана конструкция вида

RETURNS(*d*₁, *d*₂, …, *d*ₙ) ($n \geq 1$) или **ВОЗВРАЩАЕТ(*d*₁, *d*₂, …, *d*ₙ)** ($n \geq 1$)

где d_i ($i = 1, 2, ..., n$) — описатель типа вычисляемого значения. Часть описателей типа может быть опущена, они будут приписаны по общим правилам умолчания.

Если имя процедуры-функции является внутренним, то в языке PL/1 дополнительного описания значения функции в вызывающей процедуре не трбуется. Однако для процедур-функций, являющихся внешними, этот тип должен быть явно описан в вызывающей процедуре (см. § 4.3).

Что касается типа значения выражения *e* в операторе RETURN(*e*)/ВОЗВРАТ(*e*), то он может быть, вообще говоря, произвольным. Все необходимые преобразования значения этого выражения всегда выполняются автоматически.

Примеры описаний заголовков процедур-функций:

```
maxf24: proc(x) returns (float (24));
fmax53: proc(x) returns (float (53));
```

```
maxf24: проц(x) возвращает (вещ (24));
fmax53: проц(x) возвращает (вещ (53));
```

Проверь себя

1. Какими операторами может завершаться выполнение процедуры, вызванной посредством указателя функции?

2. Какое число будет напечатано при выполнении следующей программы?

```
т: proc main;                    т: проц главная;
put list(d**2);                  писать в_виде(d**2);
d: proc returns(float);          d: проц возвращает(вещ);
return(5e1); end d;              возврат(5e1); конец d;
end t;                           end t;
```

3. Составьте процедуру-функцию, вычисляющую длину отрезка на плоскости при заданных полярных координатах его вершины по формуле:

$$d = \sqrt{r_1^2 + r_2^2 - 2\,r_1\,r_2\cos(\varphi_1 - \varphi_2)}\,.$$

Используя эту процедуру, составьте программу, вводящую полярные координаты вершин треугольника и печатающую его периметр (сравните с примером в данном параграфе).

4.3. Описание имен входа в процедуры, переменные и параметры со значениями типа входа

Имеются следующие требования к описанию имен входа в процедуры в операторах DECLARE/ОПИСАНИЕ:

– в языке PL/1 имена входа во внешние процедуры всегда должны быть явно описаны в вызывающих их процедурах;

– в языке PL/1 имена входа во внутренние процедуры никогда не описываются в операторах описания;

Описание имени входа имеет вид $pa_1a_2\ldots a_n\ (n \geq 1)$, где p — имя входа; $a_i\ (i = 1, 2, \ldots, n)$ — описатель. Из описателей, допустимых для имен точек входа, здесь рассмотрим лишь описатели ENTRY/ДЛЯ_ВЫЗОВА и RETURNS/ВОЗВРАЩАЕТ.

Описатель **ENTRY/ДЛЯ_ВЫЗОВА** имеет вид

ENTRY $(d_1, d_2, \ldots, d_n)$ $(n \geq 0)$ или ДЛЯ_ВЫЗОВА$(d_1, d_2, \ldots, d_n)$ $(n \geq 0)$.

В нем d_i $(i=1, 2, \ldots, n)$ — либо пусто, либо список описателей i-го параметра. Число n (количество d_i) должно быть равно количеству параметров вызываемой процедуры. Список описателей для параметров-переменных и параметров массивов имеет вид $a_1a_2\ldots a_m$ $(m \geq 1)$, где a_i $(i=1, 2, \ldots, m)$ — описатель; для параметров-массивов a_1 должен быть описателем измерений. Часть описателей параметра может быть опущена, недостающие описатели будут при этом добавлены в соответствии с теми же правилами умолчания, что и при описании обычных переменных. Выносить общие описатели за

скобки внутри описателя ENTRY/ДЛЯ_ВЫЗОВА нельзя. В описателе измерений границы могут быть заданы только константами.

Если параметр описан в описателе ENTRY/ДЛЯ_ВЫЗОВА, то при трансляции программы проверяется, соответствует ли аргумент этому описанию параметра. При несоответствии, по возможности, создается фиктивный аргумент с требуемыми характеристиками, которому присваивается значение исходного аргумента и который фактически и сопоставляется параметру. Таким образом, в данном случае аргумент, соответствующий параметру-переменной, может быть скалярным выражением со значением любого типа, для которого допускается преобразование к требуемому типу. Аргумент, соответствующий параметру-массиву, должен быть массивом со значением той же размерности, что и у параметра, и с точно такими же типами значений элементов массива-аргумента.

Описатель **RETURNS/ВОЗВРАЩАЕТ** применяется для описания в вызывающей процедуре типа значения процедуры-функции, вызываемой посредством данного имени точки входа. Описатель имеет вид

RETURNS$(a_1 a_2 \ldots a_n)$ $(n \geq 1)$ или ВОЗВРАЩАЕТ$(a_1 a_2 \ldots a_n)$ $(n \geq 1)$

где a_i $(i = 1, 2, \ldots n)$ — описатель типа значения функции (другие описатели не допускаются). Если часть описателей опущена, то они приписываются в соответствии с общими правилами умолчания.

Имена встроенных функций в языке PL/l обычно явно не описываются. Но так как программист может применять эти же имена для обозначения переменных, массивов или, например, собственных процедур, то возможна следующая ситуация. Вне некоторой внутренней процедуры имя встроенной функции используется для обозначения какого-либо другого объекта, а внутри данной процедуры необходимо обратиться к самой встроенной функции. Тогда во внутренней процедуре следует описать имя встроенной функции с описателем **BUILTIN/РОДНАЯ**. Никакие другие описатели для имен встроенных функций недопустимы. Пример использования описателя BUILTIN/РОДНАЯ приведен в конце § 4.5.

В языке PL/1 допускаются переменные, значениями которых могут быть имена входа в процедуры. Имена этих переменных наравне с именами-константами входа в процедуры могут быть использованы и в операторах вызова процедур, и в указателях функций. Эти переменные могут входить в состав совокупностей переменных, в частности, в массивы. Таким образом, допустим указатель функции вида

$$f(i_1, i_2, \ldots, i_k)\ (a_1, a_2, \ldots, a_m)\ (k \geq 1, m \geq 0),$$

где *f* — идентификатор массива переменных типа входа; i_j *(j= 1, 2, ..., k)* — индекс элемента этого массива; a_p *(p= 1, 2, . .., m)* — аргумент.

При описании характеристик значения переменных типа входа (такое описание обязательно) им должен быть приписан хотя бы один из описателей ENTRY/ДЛЯ_ВЫЗОВА, RETURNS/ВОЗВРАЩАЕТ. Кроме того при описании простых переменных типа входа им должен быть приписан описатель **VARIABLE/КОСВЕННОЕ** (он предполагается по умолчанию, если переменной приписан хотя бы один описатель, недопустимый для констант- имен входа).

Значения переменных типа входа обычно устанавливаются с помощью оператора присваивания. В правой части такого оператора может быть указано либо имя другой переменной типа точки входа, либо константа-имя входа (но не имя встроенной функции). К значениям типа точки входа применимы операции сравнения «равно» (=) и «не равно» (^=).

Переменные типа входа могут быть параметрами. Они всегда должны быть описаны в операторе DECLARE, описатель VARIABLE при этом не требуется.

Аргументы, сопоставляемые таким параметрам, могут быть именами переменных типа входа, именами параметров типа входа, константами-именами входа (не именами встроенных функций); аргумент не может быть именем с собственным списком аргументов, так как в этом случае он будет рассматриваться как указатель функции. Для того чтобы обеспечить создание фиктивного аргумента, имя, сопоставляемое параметру типа точки входа, может быть заключено в скобки.

В качестве примера использования параметров типа входа приведем процедуру вычисления приближенного значения первой производной произвольной функции одного аргумента.

```
dif1: proc(f, x, dx);                    dif1: проц(f, x, dx);
dcl (x,dx) float;                        опс (x,dx) вещ;
dcl f entry(float) returns(float);       опс f для_вызова(вещ) возвращает (вещ);
return((f(x + dx)-f(x))/dx);             возврат((f(x + dx)-f(x))/dx);
end dif1;                                конец dif1;
```

Отметим, что процедура DIF1 применима для вычисления производной лишь тех процедур-функций, у которых аргумент и значение имеют тип FLOAT/ВЕЩ. Пусть, например, программистом составлена процедура-функция CTG (с одним параметром) и требуется напечатать значение ее производной в нуле. Для этого достаточно выполнить оператор

```
put list (dif1 (ctg, 0, 0.0001));   или   писать в_виде (dif1 (ctg, 0, 0.0001));
```

Дополнительно может потребоваться описание процедуры DIF1 в вызывающей процедуре, например, вида

```
dcl dif1 entry (entry(float) returns(float), float, float); или
опс dif1 для_вызова (для_вызова(вещ) возвращает(вещ), вещ, вещ);
```

Переменные типа входа (не параметры) обычно используются в достаточно сложных программах. В качестве упрощенного примера использования таких переменных рассмотрим программу аналитического приближения экспериментальных данных. Пусть программа вводит значения неизвестной функции *y* в точках от 0,01 до 0,99 с шагом 0,01 и пусть известно, что *y(0)=0* и *y(1)=1*. Предположим, что на языке PL/1 составлены процедуры, реализующие 10 функций f_i *(i=1, 2, ..., 10)*, таких что $f_i(0)=0$, $f_i(1)=1$, например

```
f1: proc(x); returns(float);
dcl x float;
return((exp(x) - 1)/(exp(1) - 1));
end f1;
```

```
f1: проц(x); возвращает(вещ);
опс x вещ;
возврат((exp(x) - 1)/(exp(1) - 1));
конец f1;
```

Искомая программа, печатающая номер наиболее близкой к *y* функции f_i (в смысле средне-квадратичного отклонения) может иметь вид

```
pribli: proc main;
dcl (i, j, nf) fixed(31),
y(99) float, (x, sm, sm1, sm2) float,
f(10) entry(float) returns(float),
(fl, f2, f3, f4, f5, f6, f7, f8, f9, f10)
entry (float) returns(float);
f(1) = f1; f(2) = f2;
f(3) = f3; f(4) = f4;
f(5) = f5; f(6) = f6;
f(7) = f7; f(8) = f8;
f(9) = f9; f(10) = f10;
sm = -1;
get list(y);
do i = 1 to 10;
    sm2 = 0;
    do j = 1 to 99;
       x = j/100e0;
      sm2+ =(y(j) - f(i)(x))**2;
    end j;
    if sm = - 1 | sm > sm2
     then {; nf = i; sm=sm2; };
```

```
приближение: проц главная;
опс (i, j, nf) точное(31),
(y(99), (x, sm, sm1, sm2)) вещ,
f(10)                для_вызова(вещ)
возвращает(вещ),
(fl, f2, f3, f4, f5, f6, f7, f8, f9, f10)
для_вызова (вещ) возвращает(вещ);
f(1) = f1; f(2) = f2;
f(3) = f3; f(4) = f4;
f(5) = f5; f(6) = f6;
f(7) = f7; f(8) = f8;
f(9) = f9; f(10) = f10;
sm = -1;
читать в_виде(y);
цикл i = 1 до 10;
    sm2 = 0;
    цикл j = 1 до 99;
       x = j/100e0;
      sm2 += (y(j) - f(i)(x))**2;
    конец j;
    если sm = - 1 | sm > sm2
```

```
end i;                                    тогда {; nf = i; sm=sm2; };
put list(nf);                          конец i;
end pribli;                              писать в_виде(nf);
                                          конец приближение;
```

Проверь себя

1. В каких случаях требуется описание имен входа в операторах описания?
2. Для чего используется описатель ENTRY/ДЛЯ_ВЫЗОВА?
3. В каких случаях применяется описатель BUILTIN/РОДНАЯ? Какие другие описатели могут быть приписаны именам, для которых задан описатель BUILTIN/РОДНАЯ?
4. Как устанавливаются значения переменных типа входа?
5. При каком условии оператор ВЫЗОВ A(1)(I); является синтаксически правильным?
6. Какие аргументы могут быть сопоставлены параметрами типа входа?
7. Составьте процедуру-функцию, приближенно вычисляющую значение первой производной функций одного аргумента, при условии, что значение и аргумент этих функций имеют тип ВЕЩ(53).
8. Переделайте приведенную выше процедуру ПРИБЛИЖЕНИЕ таким образом, чтобы в ней вместо обращения с элементами массива переменных типа входа использовалось бы обращение к параметру типа входа.

4.4. Сфера действия описания имен

Каждый идентификатор, используемый в программе, считается описанным либо явно, либо контекстуально, либо неявно. К явному описанию идентификатора относится, во-первых, описание его в операторе DECLARE/ОПИСАНИЕ; во-вторых, использование идентификатора в качестве метки оператора. В последнем случае, если идентификатор стоит перед оператором, отличным от PROCEDURE/ПРОЦЕДУРА или ENTRY/ДЛЯ_ВЫЗОВА, то он считается константой-меткой, если же идентификатор стоит перед оператором PROCEDURE/ПРОЦЕДУРА или ENTRY/ДЛЯ_ВЫЗОВА, то он считается именем входа в процедуру.

Идентификатор считается описанным контекстуально, если он не описан явно и использован в определенном контексте, предполагающем те или иные свойства объекта (например, обращение к встроенной функции), обозначенного идентификатором.

Описание любого идентификатора имеет ограниченную сферу действия. Обращение к идентификатору вне сферы действия какого-либо его описания всегда будет подразумевать обращение к объекту, отличному от того, который задан указанным описанием.

Сфера действия описания зависит от способа описания и от того, какой из описателей: **INTERNAL/МЕСТНОЕ** или **EXTERNAL/ОБЩЕЕ** приписан идентификатору. Ровно один из этих описателей, называемых описателями *сферы действия описания,* всегда явно или по умолчанию приписывается каждому идентификатору. Для ключевого слова EXTERNAL допустимо сокращение EXT, а для его русского эквивалента допустимы слова ОБЩАЯ или ОБЩЕЕ.

Сфера действия явного описания идентификатора, которому приписан описатель INTERNAL/МЕСТНОЕ, определяется следующим образом. Во-первых, описание действует в той процедуре, в которой непосредственно содержится данное описание (отметим, что имена входа во внутренние процедуры считаются описанными не внутри данной процедуры, а в объемлющей ее процедуре). Во-вторых, явное описание действует во всех процедурах, внутренних по отношению к той, где находится это описание, при условии, что в них не действует другое явное описание этого же идентификатора. Например, в процедуре

```
a: proc;                          a: проц;

м: proc;                          м: проц;
dcl м(10) fixed;                  опс м(10) точное;

к: proc(m);                       к: проц(m);
dcl м float;                      опс  м вещ;
…                                 …
   end к;                            конец к;
n: proc;                          n: проц;

end n;                            конец n;
end m;                            конец m;
p: proc;                          p: проц;
…                                 …
end p;                            конец p;
end a;                            конец a;
```

явное описание идентификатора *M* как имени внутренней процедуры *M* действует внутри процедуры *A* (исключая процедуру *M*) и внутри процедуры *P*. Явное описание *M* как имени массива действует внутри процедуры *M* (исключая процедуру *K*) и внутри процедуры *N*. Явное описание *M* как имени параметра-переменной действует только внутри процедуры *K*.

Сферой действия контекстуального и неявного описания идентификатора (если ему по умолчанию приписан описатель INTERNAL/МЕСТНОЕ) считается вся внешняя процедура, за исключением тех ее частей, где действует явное описание данного идентификатора.

Если имени приписан описатель EXTERNAL/ОБЩЕЕ (такие имена называются внешними), то сфера действия описания определяется следующим образом. Рассмотрим все описания данного идентификатора (во всех внешних процедурах), в которых ему приписан описатель EXTERNAL/ОБЩЕЕ, и для каждого такого описания определим сферу его действия по рассмотренным выше правилам. Объединение всех полученных сфер действия описаний и будет составлять фактическую сферу действия описания внешнего имени. Естественно, что при этом все описания идентификатора, в которых ему приписан описатель EXTERNAL/ОБЩЕЕ, должны быть идентичными.

В качестве примера рассмотрим две внешних процедуры

```
a: proc;                          a: проц;
dcl m(10) float ext;              опс m(10) вещ общее;

end a;                            конец a;

в: proc;                          в: проц;
dcl m(10) float ext;              опс m(10) вещ общее;

end b;                            конец b;
```

Здесь оба явных описания идентификатора *M* задают один и тот же массив. Если бы хотя бы в одном случае описатель EXTERNAL/ОБЩЕЕ отсутствовал, то описания массива *M* задавали бы два различных массива: один — доступный только в процедуре *A*, другой — доступный только в процедуре *B*.

Описатели сферы действия описаний могут быть явно приписаны в операторе DECLARE/ОПИСАНИЕ любым идентификаторам (кроме имен элементов структур — см. § 7.1). Если эти описатели явно не заданы, то по умолчанию именам входа во внешние процедуры приписывается описатель EXTERNAL/ОБЩЕЕ, а именам переменных, массивов и внутренних процедур — описатель INTERNAL/МЕСТНОЕ.

Учитывая рассмотренные концепции, укажем на дополнительные способы связи по информации между процедурами. Во-первых, во внутренней процедуре можно обращаться к переменным, массивам и т. д., описанным в объемлющей ее процедуре. Во-вторых, в двух внешних по отношению друг к другу процедурах можно описать, используя внешние имена, одни и те же

переменные, массивы и т. д. Однако с точки зрения безошибочности программирования оба эти способа считаются менее надежными, чем использование параметров.

П р о в е р ь с е б я

1. В каких случаях идентификатор считается описанным явно?
2. Какова будет сфера действия описания метки оператора присваивания, расположенного внутри главной процедуры?
3. Какова будет сфера действия описания метки оператора ENTRY/ДЛЯ_ВЫЗОВА, расположенного внутри процедуры, непосредственно входящей в главную процедуру?
4. Какой смысл имеют идентификаторы A, B и C внутри процедуры K?

```
n: proc(c);

a: x = 1;
b: y = 2;

dcl c(10) float;
к: proc(c);
   dcl c fixed;
к: proc(c);
b: z = 3;
dcl a entry;

end m;
end k;
end n;
```

```
n: проц(с);

a: x = 1;
b: y = 2;

опс c(10) вещ;
к: проц(с);
   опс с точное;
к: проц(с);
b: z = 3;
опс  а для_вызова;

конец m;
конец k;
конец n;
```

5. В каких случаях одна и та же переменная, может быть доступна в разных процедурах?

4.5. Автоматическое распределение памяти, блоки

Переменные и массивы с именами, не являющимися внешними, обычно недоступны вне процедуры, в которой они описаны. Более того, при завершении выполнения процедуры значения, присвоенные этим переменным или массивам, становятся ненужными. Поэтому, как правило, полезно освобождать занимаемую ими память и распределять ее под другие данные. Для того чтобы автоматически выполнять подобный процесс, в языке PL/1 предусмотрен специальный класс переменных и массивов, называемых *автоматически размещаемыми.* Все рассмотренные нами в предыдущих

параграфах и главах переменные относились именно к этому классу. Другие способы размещения данных будут рассмотрены в гл. 9.

Считается, что автоматически размещаемым переменным или массивам приписан описатель **AUTOMATIC/ВРЕМЕННОЕ**, относящийся к описателям *способа размещения.* Этот описатель может быть явно указан в операторе описания, однако в этом нет никакой необходимости, так как он приписывается по умолчанию во всех допустимых случаях. Для ключевого слова AUTOMATIC допустимо сокращение AUTO. Переменные и массивы с внешними именами ни при каких условиях не могут быть автоматически размещаемыми.

Память распределяется под автоматически размещаемые переменные и массивы в начале выполнения процедуры, в которой они описаны.

При завершении работы процедуры память, занятая описанными в ней автоматически размещаемыми переменными и массивами, освобождается и может в дальнейшем быть использована под другие данные. Отметим, что отведение и освобождение памяти под фиктивные аргументы производится так же как и для переменных с автоматическим размещением.

По определению *блок* есть последовательность операторов, начинающаяся с *оператора заголовка блока,* имеющего в простейшем случае вид **BEGIN/БЛОК** и заканчивающаяся соответствующим оператором конца. Метки перед оператором BEGIN/БЛОК могут отсутствовать.

Блоки во многом аналогичны процедурам. Во-первых, сфера действия описаний, непосредственно расположенных внутри блока, ограничивается данным блоком так же, как и в случае процедур. Во-вторых, в начале выполнения блока производятся те же действия по распределению памяти, что и в начале выполнения процедуры; то же самое относится и к освобождению памяти в момент завершения работы блока. Понятие активный блок аналогично понятию активная процедура; в частности, никаким способом нельзя передать управление внутрь блока, не являющегося активным. Основные отличия блока от процедуры состоят в следующем:

а) Вызов блока осуществляется только посредством обычной передачи управления оператору его начала, т. е. либо с помощью оператора перехода, указывающего метку оператора BEGIN/БЛОК, либо после выполнения оператора, предшествующего оператору BEGIN/БЛОК (следовательно, в данном случае расположение блока внутри объемлющей процедуры не является произвольным).

б) Возврат управления из блока осуществляется либо посредством оператора перехода, передающего управление во вне блока, либо после

выполнения оператора конца блока. В последнем случае далее будет выполняться оператор, следующий за оператором конца.

в) Любой блок должен быть частью какой-либо процедуры или другого блока.

г) Для блока не могут быть заданы параметры.

Рассмотрим один специфический пример использования блока. Предположим, что программист составил свою собственную процедуру-функцию для вычисления синуса и назвал ее SIN, т. е. так же как и соответствующую встроенную функцию (что во многих отношениях оправдано). Теперь необходимо составить программу, которая бы для углов 0.1, 0.2, 0.3, ..., 0.9, 1 печатала разность между значениями синуса, вычисленными с помощью встроенной функции и процедуры-функции, составленной программистом. Эта задача может быть решена с помощью программы

```
sinsin: proc main;
dcl (s, x) float;
do x = 0.1 to 1;
s = sin(x);
begin;
dcl sin builtin;
put list(s-sin(x));
end;
end x;
sin: proc(x);
... <процедура-функция для
вычисления синуса>
end sin;
end sinsin;
```

```
sinsin: проц главная;
опс (s, x) вещ;
цикл x = 0.1 до 1;
s = sin(x);
блок;
опс sin родная;
писать в_виде(s-sin(x));
конец;
конец x;
sin: проц(x);
... <процедура-функция для
вычисления синуса>
конец sin;
конец sinsin;
```

Здесь блок необходим для того, чтобы разделить сферы действия двух явных описаний идентификатора SIN. Этот же пример иллюстрирует возможность использования описателя BUILTIN/РОДНАЯ.

Проверь себя

1. В какой момент распределяется память под значения автоматически размещаемых переменных и когда она освобождается?
2. Какими способами можно передать управление блоку?
3. При выполнении каких операторов может быть закончено выполнение блока?

4. Назовите какие-либо ситуации, когда полезно использовать блоки?

4.6. Трансляция и выполнение программы из нескольких внешних процедур

Трансляция и выполнение программы, состоящей из нескольких внешних процедур, может в простейшем случае иметь точно такой же вид, как и для программы, состоящей из одной процедуры. Сначала транслируются тексты всех внешних процедур.

Например, пусть необходимо составить программу для вычисления и печати значения величины

$\sum_{i=1}^{N} C_{3N-i}^{i+1}$, где $C_n^m -$ число сочетаний из n элементов по m.

Программу составим из трех внешних процедур. Процедура FAC, вычисляющая значение функции *n*! (факториал от *n),* может иметь вид FAC:

```
fac: proc(n) returns (bin(31));
dcl (n, i, s) binary(31);
s= 1;
do i = 2 to n; s = s * i; end i;
return(s); end fac;
```

```
fac: проц(n) возвращает (двоичное(31));
опс (n, i, s) двоичное(31);
s= 1;
цикл i = 2 до n; s = s * i; конец i;
return(s); end fac;
```

Процедура COMB, вычисляющая значение числа сочетаний C_n^m по формуле

$C_n^m = \frac{n!}{m!(n-m)!}$, может иметь вид

```
comb: proc(n, m) returns(bin
(31));
dcl (n,m) bin(31);
dcl fac entry(bin(31))
returns(bin(31));
return(fac(n)/(fac(m) * fac(n-
m)));
end; comb
```

```
comb: проц(n, m)
возвращает(двоичное (31));
опс (n,m) двоичное(31);
опс fac для_вызова(двоичное(31))
возвращает(двоичное(31)) ;
возврат(fac(n)/(fac(m) * fac(n- m)));
конец comb;
```

Главная процедура, вычисляющая и печатающая значение указанной выше суммы, может иметь вид

```
man: proc main;
dcl (n, i, s) bin(31);
dcl comb entry(bin(31), bin (31))
returns(bin(31));
get list(n);
s = 0;
```

```
man: проц главная;
опс (n, i, s) двоичное(31);
опс comb для_вызова(двоичное(31),
двоичное (31))
возвращает(двоичное(31));
читать в_виде(n);
```

```
do i= 1 to n;
    s += comb(3*n-i,i + l);
end i;
put list(s);
end man;
```

```
s = 0;
цикл i= 1 до n;
    s +=  comb(3*n-i,i + l);
конец i; писать в_виде (s);
конец man;
```

Трансляция и выполнение программы, состоящее из приведенных выше внешних процедур MAN, COMB и FAC, при использовании компилятора PL/1 может иметь вид директив командной строки:

```
PLINK64 MAN.PL1
PLINK64 COMB.PL1
PLINK64 FAC.PL1
PLINK64 MAN.OBJ, COMB.OBJ, FAC.OBJ
```

или (так как расширение по умолчанию подставляется .OBJ)

```
PLINK64 MAN, COMB, FAC
```

Запуск полученной программы MAN.EXE – директивы командной строки:

```
MAN.EXE
```

Мы рассмотрели вопрос о том, как транслировать и выполнять программу, состоящую из нескольких внешних процедур, тексты которых имеются в распоряжении программиста. Однако на практике часто приходится использовать внешние процедуры, оттранслированные отдельно и помещенные в библиотеки объектных модулей (понятие библиотеки модулей будет подробно разъяснено в гл. 11). В частности, имеются библиотеки стандартных подпрограмм, предназначенных, например, для обработки матриц, для численного интегрирования, для решения всевозможных систем уравнений и т.д. Естественно, если имеются подобные библиотеки, то, как правило, не имеет смысла составлять собственные процедуры для решения этих, зачастую весьма сложных, задач.

Зная имена и требования к аргументам стандартных подпрограмм, необходимых программисту, он может обращаться к ним по тем же правилам, что и к своим собственным внешним процедурам. Однако при этом необходимо присоединить вызываемые подпрограммы к программе, составленной самим программистом.

В простейшем случае в директиве командной строки вместо или вместе с именами оттранслированных собственных внешних процедур перечисляются и имена библиотек объектных модулей, которые всегда имеют расширение .L86. Например, если бы приведенная выше программа FAC уже входила бы объектным модулем в библиотеку MATH.L86, то трансляция внешних

процедур и получение выполняемой программы MAN свелось бы к директивам командной строки

```
PLINK64 MAN.PL1
PLINK64 COMB.PL1
PLINK64 MAN.OBJ, COMB.OBJ, MATH.L86
```

Проверь себя

1. Зависит ли трансляция каждой внешней процедуры от других внешних процедур?
2. Какой директивой командной строки можно получить выполняемую программу, состоящую из трех внешних процедур F1, F2, F3 и библиотеки математических функций INTEGRAL.L86?
3. Можно ли в директиве получения выполняемой программы указывать несколько библиотек объектных модулей?

5. РАБОТА СО СТРОКАМИ

5.1. Переменные со значениями типа битовых строк

Минимальной единицей информации, хранимой в памяти, является *бит* (сокращение от binary digit – двоичная цифра). Каждому биту соответствует одно из двух значений, обычно обозначаемых 0 или 1. Любые данные, хранимые в памяти, всегда представляются в виде последовательности или *строки битов.* Например, значение переменной типа FIXED BINARY(15) или ТОЧНОЕ ДВОИЧНОЕ(15), равное -1, представляется строкой битов 111111111111111, а значение переменной типа FIXED DECIMAL(1, 0) или ТОЧНОЕ ДЕСЯТИЧНОЕ(1,0), равное 5, представляется строкой битов 00000101.

В языке PL/1 имеется возможность хранить, сравнивать и выполнять ряд преобразований над данными типа строк битов (или *битовых строк).* Для изображения конкретных битовых строк применяются *константы,* имеющие вид строк в кавычках

$'b_1b_2 \ldots b_n'B$ либо $'b_1b_2 \ldots b_n'\,(k)B$ (n ≥ 0).

Здесь $b_i\,(i = 1, 2, \ldots, n)$ цифры 0 или 1; k — целое положительное десятичное число, называемое *повторителем* константы. Пробелы и подчеркивание внутри кавычек игнорируются, но перед буквой ***B*** или ее эквивалентом ***Б*** недопустимы. Значением константы без повторителя является битовая строка $b_1b_2\ldots b_n$. Например, значением константы '10'B является строка из двух битов 10. Значением константы с повторителем является строка, состоящая из $n{\bullet}k$ битов

$\underline{b_1b_2\ldots b_n}$	$\underline{b_1b_2\ldots b_n}$	$\underline{b_1b_2\ldots b_n}$
1-я подстрока	2-я подстрока	*k*-я подстрока

Например, значением константы '101'(3)B является строка 101101101.

Количество битов в строке называется *длиной* строки. В данной версии языка битовые строки нулевой длины (пустые), которые изображаются константой "B не допускаются.

Имеются разновидности констант *B:* ***B1***, ***B2***, ***B3*** и ***B4*** (русские эквиваленты ***Б1***, ***Б2***, ***Б3*** и ***Б4***), которые обозначают битовые значения в разных системах счисления, а именно в двоичной, в четверичной, восьмеричной и шестнадцатеричной.

Обозначение констант *B/Б* и *B1/Б* эквивалентны. Для вида *B2/Б2* допустимы символы 0, 1, 2, 3. Для вида *B3/Б3* допустимы символы 0, 1, 2, 3, 4,

5, 6 и 7. Для вида *B4/Б4* допустимыми являются символы 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. Таким образом, если битово-строчная константа длиной 16 содержит значение из всех единиц, то ее эквивалентные обозначения в разных видах: '1'(16)B или '1111111111111111'B или '33333333'B2 или '177777'B3 или 'FFFF'B4.

Для манипулирования данными типа строк битов, так же как и для манипулирования числовыми данными, используются переменные. У таких переменных в данной версии языка все значения имеют одинаковую или постоянную длину. Тип их значений задается с помощью описателя **BIT** или **БИТ** (всегда должен быть указан) и описателя длины значений вида *(n)*, где

$$1 \leq n \leq 64$$

Описатель длины битово-строчных переменных всегда должен непосредственно следовать за описателем **BIT/БИТ**. Если описатель длины опущен, то предполагается, что он имеет вид (1).

В качестве примера рассмотрим оператор описания

dcl a bit (2); или опс a бит(2);

Здесь описана битово-строчная переменная, значениями которой могут быть строки, изображаемые константами '00'Б, '01'Б, '10'Б и '11'Б.

Битово-строчные переменные могут быть объединены в массивы. Например, в операторе описания

dcl x (3) bit (5), y (6,6) bit (8); опс x (3) бит (5), y (6,6) бит (8);

описаны два массива битово-строчных переменных. Значения всех переменных, входящих в массив X, имеют длину, равную 5 битам; значения всех переменных, входящих в массив Y, имеют длину, равную 8 битов.

Установка значений битово-строчных переменных осуществляется с помощью тех же операторов языка PL/1, что и в случае числовых переменных. При этом если длина присваиваемой строки битов больше длины, допустимой для данной переменной, то часть правых битов строки отбрасывается. Например, пусть описаны переменные

dcl a bit(3), b bit (4); опс a бит(3), b бит(4);

После выполнения оператора A,B='10101'Б; значения переменных A и B будут равны соответственно '101'Б и '1010'Б. Если длина присваиваемой строки битов меньше заданной для переменной, то строка дополняется справа нулевыми битами.

В правой части оператора присваивания, устанавливающего значения битово-строчных переменных, может стоять выражение со значением, отличным от строки битов, в частности с числовым значением. Число преобразуется в строку битов по следующим правилам: если число не целое,

то у него отбрасывается дробная часть, если число отрицательное, то в дальнейшем используется абсолютная величина его значения; представление полученного вещественного целого неотрицательного числа в двоичной системе счисления с помощью n цифр и рассматривается как искомая строка из n битов. Длина строки n зависит от характеристик исходного числа; для десятичных чисел в форме с плавающей точкой и точностью (p)

$$n= min(31, [3, 32p]);$$

для двоичных чисел в форме с плавающей точкой и точностью (p) n равно $min(63, p)$, для десятичных чисел в форме с фиксированной точкой и точностью (p, q)

$$n=min(max([3,32(p-q)], 0), 31);$$

На практике преобразование чисел в строку битов используется редко. Исключением может служить оператор вида $u=0;$ где u — переменная с битово-строчным значением. При выполнении этого оператора переменной v будет всегда присвоена строка из нулевых битов требуемой длины. Однако ненулевые числовые значения можно присваивать битово-строчным переменным, только твердо зная правила преобразования, изложенные выше. Значения битово-строчных переменных могут быть установлены оператором GET/ЧИТАТЬ наряду со значениями числовых переменных. Вводимые значения при этом задаются в виде битовых констант (впрочем, могут быть использованы и другие константы, преобразуемые в битовые строки).

Одним из основных случаев, когда программист имеет дело с битовыми строками, является вычисление логических выражений. Значение любой операции сравнения есть строка из одного бита, равная '1'Б, если операнды удовлетворяют сравнению и '0'Б в противном случае. В качестве простейшего примера использования битово-строчных переменных рассмотрим программу, вычисляющую значения функций

$$f_1(x)=\begin{cases} x^2 - x & \text{при } 0 \le x \le 1 \\ 2x^2 - x - 1 & \text{в остальных случаях} \end{cases}$$

$$f_2(x)=sin^2 f_1(x)+ cos^2 f_1(x)+$$

$$f_3(x)=\begin{cases} f_2(x) - 1 & \text{при } 0 \le x \le 1 \\ x^3 - f_2(x) & \text{в остальных случаях} \end{cases}$$

Учитывая, что при вычислении необходимо дважды проверять условие $0 \le x \le 1$, составим программу следующим образом:

```
f123: proc main;                    f123: проц главная;
dcl (x,f1,f2,f3) float;             опс (x,f1,f2,f3) вещ;
get list(x);                        читать в_виде(x);
```

```
dcl b bit(l);
b = 0 < = x & x  < = 1 ;
/*b = '0'b при x<0 и x> 1*/
if b then f1 = x ** 2-x; else
          f1 = 2 * x **2- x - 1;
f2 = sin(f1) ** 2 + cos(f1) ** 2;
if b then f3 = f2-1; else
          f3 = x ** 3 - f2;
put list(f1, f2, f3);
end;
```

```
опс b бит(l);
b = 0 < = x И x  < = 1 ;
/*b = '0'b при x<0 и x> 1*/
если b тогда f1 = x ** 2-x; иначе
          f1 = 2 * x **2- x - 1;
f2 = sin(f1) ** 2 + cos(f1) ** 2;
если b тогда f3 = f2-1; иначе
          f3 = x ** 3 - f2;
писать в_виде(f1, f2, f3);
конец;
```

П р о в е р ь с е б я

1. Могут ли битово-строчные константы содержать что-либо, кроме 0 и 1?
2. Составьте оператор описания двух битово-строчных переменных: одной со значениями длиной 5 битов, другой со значениями длиной 64 бита.
3. Составьте оператор описания массива битово-строчных переменных со значениями длиной 32 бита.
4. Используя битовые переменные, составьте программу для вычисления функции

$$f(x,y)=\begin{cases} 1 \text{ при } 1 \le x \le 2 & y > 2 \\ 2 \text{ при } 1 \le x \le 2 & 1 \le y \le 2 \\ 3 \text{ при } x > 2 & 1 \le y \le 2 \end{cases}$$

5.2. Операции над строками битов, встроенные функции для строк битов, правила вычислений логических выражений

К битовым строкам любой длины (и только к ним) применимы операции *логического отрицания (^), логического умножения* (&) и *логического сложения* (|). Русские эквиваленты этих операций НЕ, И, ИЛИ соответственно. Ниже приведены результаты применения этих операций к строкам единичной длины:

```
^ '0'Б='1'Б                ^ '1'Б = '0'Б
'0'Б & '0'Б = '0'Б         '0'Б & '1'Б = '0'Б
'1'Б & '0'Б = '0'Б         '1'Б & '1'Б = '1'Б
'0'Б  |  '0'Б = '0'Б       '0'Б  |  '1'Б = '1'Б
'1'Б  |  '0'Б = '1'Б       '1'Б  |  '1'Б = '1'Б
```

Или с русскими эквивалентами операций

НЕ '0'Б='1'Б НЕ '1'Б = '0'Б

'0'Б И '0'Б = '0'Б '0'Б И '1'Б = '0'Б

'1'Б И '0'Б = '0'Б '1'Б И '1'Б = '1'Б

'0'Б ИЛИ '0'Б = '0'Б '0'Б ИЛИ '1'Б = '1'Б

'1'Б ИЛИ '0'Б = '1'Б '1'Б ИЛИ '1'Б = '1'Б

Если длина операндов не равна одному биту, то *i*-й бит результирующей строки равен результату применения операции, к *i*-m битам строк-операндов. Например:

^'10'Б = '01'Б;

'101'Б & '110'Б = '100'Б;

'010'Б | '100'Б = '110'Б;

Если в случае двуместной операции длины строк различны, то меньшая строка считается дополненной справа нулевыми битами. Например,

'11'Б&'011'Б = '010'Б, '1'Б | '010'Б ='110'Б.

Если один или оба операнда любой из трех логических операций не являются строкой битов, то операнд преобразуется в строку битов (см. § 5.1 и § 5.4).

К битовым строкам применимы все операции сравнения. При этом если сравниваются строки различной длины, то меньшая считается дополненной справа нулевыми битами. Результат сравнения определяется на основе попарного сравнения соответствующих битов обеих строк. Причем строка $b_L b_2 . . , b_n$ больше чем строка $c_1 c_2 \ldots c_n$ в том случае, если для некоторого $k\ (1 \le k \le n)$ выполняются условия

$b_1 = c_1\ b_2 = c_2, \ldots, b_{k-1} = c_{k-1}\ b_k = '1'Б\ c_n = '0'Б$. Например, следующие логические выражения истинны:

'00'Б = '0'Б; '01'Б ^ = 'l'Б; '101'Б = '1010'Б;

'10'Б < '11'Б '0'Б < '1'Б '0111'Б < '10'Б

а следующие логические выражения ложны:

'01'Б = '1'Б '10'Б ^ = '1'Б;

'101100'Б > '1100'Б; '11'Б < '10111'Б.

Если один из операндов операции сравнения имеет числовые значения, а другой — значения типа строки битов, то строка битов преобразуется в вещественное двоичное целое неотрицательное число с точностью (63). При этом преобразовании бит 1 рассматривается как двоичная цифра 1, а бит 0 — как цифра 0.

Еще одной операцией, применимой к битовым строкам, является операция *сцепления*, обозначаемая знаком ||. Результат этой операции образуется присоединением второй строки справа к первой строке. Например:

'101'Б || '01'Б = '10110'Б '00'Б || '1'Б = '001'Б

Операция сцепления имеет 4-й приоритет. Следовательно, в следующем логическом выражении скобки могут быть опущены:

(A || '00'Б) ^ = ('1'Б || C) эквивалентно A || '00'Б ^ = '1'Б || C

Выражения с битово-строчными значениями могут быть указаны в качестве операндов арифметических операций; в этом случае перед выполнением операции битовые строки преобразуются в целые двоичные числа, так же как и в случае операций сравнения. Если битовая строка присваивается переменной с числовыми значениями в форме с фиксированной точкой, то она преобразуется предварительно по рассмотренным выше правилам в целое двоичное число. Если же битовая строка присваивается переменной с числовыми значениями в форме с плавающей точкой, то она также рассматривается как представление целого неотрицательного числа в двоичной системе счисления. Однако в данном случае максимальное допустимое количество цифр в таком числе равно 53; строки, состоящие более чем из 53 битов, усекаются слева.

При работе с битовыми строками не единичной длины часто используется встроенная функция **SUBSTR/ПОДСТРОКА(*b, k, l*),** где *b* — выражение со значением типа строки битов; *k* и *l* — выражения с целочисленными значениями. В рассматриваемой версии языка *l* может быть только константой. Результатом функции SUBSTR/ПОДСТРОКА*(b, k, l)* является строка битов, равная подстроке, начинающейся с *k*-го бита строки *b* и имеющая длину *l* битов. Например,

SUBSTR/ПОДСТРОКА('101101'Б, 2, 3)='011'Б.

Обращение к этой встроенной функции может также иметь вид SUBSTR/ПОДСТРОКА(*b, k*); тогда значением функции будет подстрока, начинающаяся с *k*-го бита строки *b* и включающая весь остаток этой строки. Например, SUBSTR/ПОДСТРОКА('101101'Б, 2)='01101'Б.

Если программисту требуется изменить часть битовой строки, то он может применить операторы присваивания вида

1) SUBSTR/ПОДСТРОКА (*u, k ,l*)=*e;* 2) SUBSTR/ПОДСТРОКА(*u, k*)=*e;*

где *u* — имя переменной, у которой требуется изменить часть ее текущего строкового значения. Аргументы *k* и *l* задают положение изменяемой подстроки так же, как и в случае встроенной функции SUBSTR/ПОДСТРОКА.

Конструкция SUBSTR/ПОДСТРОКА(*u, k, l)* или SUBSTR/ПОДСТРОКА(*u, k),* использованная в контексте, предполагающем присваивание значения, называется *псевдопеременной* SUBSTR/ПОДСТРОКА. В языке PL/1 имеется несколько псевдопеременных и каждая из них задает свой специфический способ присваивания, отличный от обычного присваивания значений переменных. Помимо левых частей операторов присваивания, псевдопеременные могут быть использованы в списках оператора ввода (GET/ЧИТАТЬ), на месте переменной, управляющей циклом, в операторах заголовка циклических групп, а также в некоторых других конструкциях языка, рассмотренных в последующих главах.

При использовании псевдопеременной SUBSTR/ПОДСТРОКА присваиваемое значение всегда преобразуется к длине изменяемой подстроки (усекается справа или дополняется справа нулевыми битами). Пусть, например, имеются переменные с битово-строчными значениями, описанные оператором

dcl a bit(5); опс a бит(5);

и пусть текущее значение этой переменной равно, '10101'Б. После выполнения оператора присваивания SUBSTR(A, 3, 3) = '01'Б; значение переменной A станет равным '10010'Б. Отметим, что при использовании псевдопеременной SUBSTR/ПОДСТРОКА никогда не изменяется длина текущего значения переменной.

При формировании логических выражений можно использовать массивы.

Пусть, например, X — числовой массив любой размерности. Тогда логическое выражение X=0 истинно в том и только в том случае, когда значения всех элементов массива равны нулю; логическое выражение X ^= 0 истинно в том и только в том случае, когда хотя бы у одного элемента массива значение не равно нулю.

Другой пример: пусть A и B — два массива с одинаковой размерностью и одинаковой индексацией. Тогда логическое выражение A=B истинно в том и только в том случае, когда все элементы массивов побайтно совпадают.

Используя псевдопеременные, следует учитывать общее ограничение, присущее версии языка PL/l, которое заключается в том, что псевдопеременная не может быть аргументом псевдопеременной.

Для анализа строк битов (например, для определения номера первого слева единичного или нулевого бита) может быть использована встроенная функция **INDEX/ИСКАТЬ**. Значением функции INDEX(*b_1, b_2)* или ИСКАТЬ*(b_1, b_2),* где b_1, и b_2 — выражения со значениями b'_1 и b'_2 типа строки битов, является номер самого левого бита b'_1, начиная с которого

строка b'_2 целиком входит в строку b'_1. Если же строка b'_2 не входит в строку b'_1, то значение функции INDEX(b_1, b_2) равно нулю. Например: значение INDEX('11100100'Б, '0' Б) равно 4, а значение INDEX ('101'Б, '11'Б) равно нулю. Значение функции INDEX/ИСКАТЬ является вещественным двоичным числом в форме с фиксированной точкой с точностью (15).

Номер первого слева единичного или нулевого бита может быть определен также с помощью встроенной функции **VERIFY/ ИСКАТЬ_НЕСОВПАДЕНИЕ**. Формально значением функции VERIFY(b_1, b_2), где b_1 и b_2 — выражения с битово-строчными значениями b'_1 и b'_2 является номер левого бита строки b'_1 не входящего в строку b'_2.

Если строка b'_1 состоит из тех же битов, что и строка b'_2, то значение функции VERIFY(b_1, b_2) равно нулю. Например, значение VERIFY ('0011'Б, '0'Б) равно 3, а значение VERIFY('11'Б, '1'Б) равно нулю. Значение функции VERIFY/ ИСКАТЬ_НЕСОВПАДЕНИЕ является вещественным двоичным числом в форме с фиксированной точкой с точностью (15).

Встроенная функция **BOOL(b_1, b_2, b_3)**, или **БУЛЕВА_АЛГЕБРА(b_1, b_2, b_3)**, где b_1, b_2 и b_3 — выражения с битово-строчными значениями, применяется для реализации произвольных булевых (логических) операций. Операндами операции являются аргументы b_1 и b_2, а сама операция задается аргументом b_3, значения которого должны быть равны строке из 4 битов (иные значения преобразуются к таковой строке); обозначим биты строки b_3 через b_3^1, b_3^2, b_3^3 b_3^4. Тогда результат применения операции (обозначим ее знаком θ) к паре соответствующих битов операндов может быть представлен формулами

'0'Б θ '0'Б=b_3^1 *'0'Б θ '1'Б* = b_3^2 *'1'Б θ '0'Б* = b_3^3 *'1'Б θ '1'Б* = b_3^4

Например, БУЛЕВА_АЛГЕБРА(b_1,b_2, '0111'Б) = b_1 *ИЛИ* b_2

Значением встроенной функции **BIT(e, k)** или **БИТ(e, k)**, где e — выражение со значением любого типа, k — целая десятичная константа, является результат преобразования значения e к битовой строке, дополненный справа нулевыми битами или усеченный справа до длины, равной k битам. При преобразовании применяются стандартные правила (см. § 5.1, § 5.4). Функция BIT/БИТ может быть использована при работе только с битовыми строками. В качестве примера приведем три оператора, каждый из которых присваивает переменной A (типа БИТ(32)) строку из 32 битов, первый из которых равен единице, а остальные нулю.

```
A = '10000000000000000000000000000000'Б;
A = '1'Б ||'0'(31)Б;
```

```
A = БИТ('1'Б, 32);
```

Значение встроенной функции **LENGTH(*e*),** или **ДЛИНА(*e*),** где *e* — скалярное выражение со значением битово-строчного типа, равно длине (в битах) значения *e*. Например, после выполнения операторов

```
dcl a bit(10),b bit(5);                опс a бит(10),b бит(5);
put list(length(a||b));                писать в_виде(длина(a||b));
```

будет напечатано число 15. Значение функции LENGTH/ДЛИНА имеет тип ТОЧНОЕ(15).

Помимо перечисленных выше при работе с битовыми строками применяются встроенная функция и псевдопеременная **UNSPEC/МАШ_КОД** (см. § 13.5).

Также для битовых строк можно использовать функцию **ROUND/ОКРУГЛИТЬ**, однако в отличие от числовых значений эта встроенная функция не округляет, а сдвигает битовые строки «по кругу» на число бит, указанное как второй параметр, который может быть числовой константой со знаком. Если указан знак «минус» — сдвиг идет влево, иначе вправо. Например, если *B* битовая строка длиной 16, то поменять в ней байты местами можно операторами (в данном случае сдвиг может быть и влево и вправо): B=ROUND(B,8); или B=ОКРУГЛИТЬ(B,-8);

Рассмотрим теперь общие правила вычисления логических выражений. Значение любого выражения, использованного в качестве логического, всегда при необходимости преобразуется в строку битов. Далее, логическое выражение считается истинным, если в его значении есть хотя бы один единичный бит. Если же значение выражения является строкой, состоящей только из нулевых битов, то выражение считается ложным.

Для иллюстрации изложенных возможностей обработки битовых строк воспользуемся следующей задачей. Пусть необходимо ввести данные о большой серии экспериментов, причем для каждого эксперимента вводится 20 чисел. Однако для последующего анализа экспериментов нет необходимости хранить эти числа, требуется лишь запомнить, имелись ли среди данных конкретного эксперимента положительные числа, отрицательные числа, нули. Такую информацию компактнее всего хранить в виде строк битов, по 3 бита на эксперимент— 1-й бит равен единице при наличии положительных чисел, 2-й — при наличии отрицательных чисел, 3-й — при наличии нулей. Рассмотрим различные подходы к решению данной задачи, причем предположим, что количество экспериментов заведомо меньше 30000.

Предположим вначале, что для хранения данных использован двумерный массив битовых строк единичной длины. Тогда ввод экспериментальных данных может быть реализован последовательностью операторов вида

```
dcl xap(30000, 3) bit(1), x(20) float;
on endfile(sysin) goto me;
do i = 1 by 1;
    get list(x);
    do j=lbound(x) to hbound(x);
       xap(i,1) = xap(i,1) | x(j) > 0;
       xap(i,2) = xap(i,2) | x(j) < 0;
     end j
     xap(i, 3) = (x =0);
end i;
me: n -= 1;
/* n — число экспериментов */
```

```
опс xap(30000, 3) бит(1), x(20) вещ;
когда конец_файла(стд_ввод) идти me;
цикл i = 1 с_шагом 1;
    читать в_виде(x);
    цикл j=ниж_граница(x) до
            верх_граница(x);
       xap(i,1) = xap(i,1) | x(j) > 0;
       xap(i,2) = xap(i,2) | x(j) < 0;
     конец j
     xap(i, 3) = (x =0);
конец i;
me: n -= 1;
/* n — число экспериментов */
```

П р о в е р ь с е б я

1. Какую операцию обозначает каждый из знаков ^, & и | ?
2. Пусть значения битово-строчных переменных A и B равны, соответственно, '11'Б и '011'Б. Вычислите значения следующих выражений:

1) ^B; 2) A | B; 3) A&B; 4) ^(A&B); 5) ^A&B;

3. Пусть значения битово-строчных переменных A и B равны соответственно '01'Б и '111'Б. Вычислите значения следующих выражений:

1) A='1'Б;
2) A < '011'Б;
3) B = '1110'Б;
4) B > '1101'Б;
5) A^= '010'Б;
6) A >= '010'Б;
7) B^ = '11'Б;
8) B <= '01111'Б.

4. Пусть значения битово-строчных переменных A и B равны, соответственно, '01'Б и '101'Б. Вычислите значения следующих выражений:

1) A||B; 2) B||A; 3) A||'0'Б||B.

5. Пусть значение битово-строчной переменной A равно '1101010'Б. Вычислите значения следующих указателей функций:

1) SUBSTR (A, 2, 1); 2) SUBSTR (A, 1, 4); 3) ПОДСТРОКА(A, 5);

6. Пусть значение переменной X является строкой битов. Составьте последовательность операторов с использованием встроенной функции INDEX/ИСКАТЬ, которая бы печатала номер первого слева единичного бита

значения X, после которого следует подряд, по крайней мере, 7 единичных битов.

7. Каково значение указателя функции БИТ('11'Б, 10)?

8. Пусть значения переменных A, B, C равны, соответственно, '01'Б, '10'Б и '00'Б. Укажите, какие из следующих выражений, рассматриваемых как логические, являются истинными.

1) A|B|C; 2) (A&B)||C; 3) A||B||C; 4) (A&C)|B.

5.3. Строки символов, символьно-строчные переменные

Вся память, как оперативная, так и на внешних запоминающих устройствах, состоит из целого числа единиц, называемых *байтами.* Последовательности произвольных байтов являются еще одним видом данных, которыми может манипулировать программа на языке PL/1. В связи с тем, что внутреннее представление символа (например, из «кириллицы Windows») обычно занимает один байт, а также в связи с тем, что последовательности байтов применяются в языке PL/1 чаще всего для изображения текстов, данные этого типа принято называть *строками символов* или *символьными строками.*

Конкретные символьные строки изображаются в программе в виде *символьных констант взятых в апострофы*

$$'c_1c_2\ldots c_n' \quad \text{или} \quad 'c_1c_2\ldots c_n'(k) \quad (n \geq 0)$$

где c_i $(i = 1, 2, .., n)$ — любой символ, который может быть представлен в памяти (кроме апострофа), либо пара апострофов; в последнем случае c_i изображается в программе как два отдельных символа; k — целое десятичное число без знака, называемое *повторителем* строковой константы.

Значение, изображаемое константой без повторителя, равно строке символов $c'_1\ c'_2\ldots\ c'_n$ где $c_i = c'_i$ $(i=1, 2, \ldots, n)$, если c_i не равно паре апострофов; если же c_i является парой апострофов, то c'_i является одним апострофом. Значение константы с повторителем равно повторенной k раз строке $c_1c_2\ldots\ c_n$. Например, значением константы 'ОБ"ЕКТ' является строка символов ОБ'ЕКТ, а значением константы 'ХА'(3) является строка ХАХАХА.

Количество символов в строке называется ее *длиной.* Длина любой строки не должна превышать 254 символа. Допускаются символьные строки нулевой длины (пустые); они изображаются константой ''.

Для манипулирования данными типа строк символов, так же как и для манипулирования числами или битовыми строками, используются переменные. Здесь различают два класса символьно-строчных переменных: первые — со значениями одинаковой постоянной длины; вторые — со

значениями произвольной длины, не превышающей некоторого предела, т. е. со значениями изменяющейся длины. Символьно-строчные переменные описываются в операторе DECLARE/ОПИСАНИЕ. Их значение характеризуется описателями **CHARACTER/ТЕКСТ** (сокращенно **CHAR**; всегда должен быть указан); **VARYING/РАЗНОЙ_ДЛИНЫ** (сокращенно **VAR/РД**; указывается в том и только в том случае, если значения переменной имеют изменяющуюся длину);

(l) — описатель длины значений, аналогичный описателю длины битово-строчных переменных, где *l* — константа, либо задающая длину каждого значения переменной (для значений постоянной длины), либо задающая максимальную допустимую длину значений переменной (для значений изменяющейся длины). Описатель длины символьно-строчных переменных всегда должен непосредственно следовать за описателем CHARACTER/ТЕКСТ. Если описатель длины опущен, то предполагается, что он равен (1). В качестве примера рассмотрим оператор описания

dcl (a, b var) char (8); опс (a, b рд) текст (8);

Здесь описаны две символьно-строчных переменных. Значением переменной A может быть любая строка, состоящая из восьми символов, например, '01234567' или 'СТРОКА--' (в дальнейшем строки символов будем как правило изображать в виде соответствующей константы языка PL/l). Значением переменной B может быть любая строка, состоящая не более чем из 8 символов, например '98764532' или 'XYZ' или пусто ''.

Символьно-строчные переменные, так же как и переменные любых других типов, могут быть объединены в массивы. Например, в операторе

dcl x(3)char(2), y(6,6) char (80) var; опс x(3)текст(2), y(6,6) текст (80) рд;

Описаны два массива символьно-строчных переменных. Значения всех переменных, входящих в массив X, имеют длину, равную 2 символам. Значения всех переменных, входящих в массив Y, имеют длину, не превышающую 80 символов.

Установка значений символьно-строчных переменных выполняется теми же операторами, что и в случае числовых или битово-строчных переменных (более того, для установки значений символьно-строчных переменных имеются специальные операторы, не применимые к переменным других типов). Если длина присваиваемой строки символов больше длины, допустимой для данной переменной, то часть правых символов присваиваемой строки отбрасывается. Если длина присваиваемой строки символов меньше требуемой для переменной со значениями постоянной длины, то присваиваемая строка дополняется справа необходимым количеством пробелов. В случае переменных со значениями изменяющейся

длины никакого дополнения присваиваемого значения не производится. Рассмотрим гримеры. Пусть описаны переменные

dcl (a, b var) char (2), (c, d var) char (4); опс (a, b рд) текст(2), (c, d рд) текст(4);

После выполнения оператора A, B, C, D='ABC'; значением переменных A и B станет 'AB', значением переменной C — 'ABC␣', переменной D — 'ABC'.

Значение, присваиваемое символьно-строчной переменной, например, оператором присваивания или оператором ввода (GET/ЧИТАТЬ), не обязано быть строкой символов. В этом случае такое значение перед присваиванием автоматически преобразуется в строку символов. При этом битовые строки преобразуются в строки, состоящие из символов 0 и 1, следующих в том же порядке и в том же количестве, что и биты 0 или 1 исходной строки. Десятичные действительные числа преобразуются в строку символов, имеющую обычно вид числовой константы, по правилам, приведенным в табл. 5.1.

ТАБЛИЦА 5.1

Результат преобразования десятичных чисел в строку символов

Форма представления и точность числа	Вид результата преобразования	Длина результата
FLOAT(*p*) ВЕЩ(*p*)	$sd_1\ d_2 \ldots\ d_p E s' d_{p+1} d_{p+2}$, *где* $x = s d_1, d_2 \ldots\ d_p\ 10^{S' d_{p+1} d_{p+2}}$ при $x \neq 0$ всегда $d_1 \neq 0$ (из этого условия определяется значение порядка)	$p + 6$
FIXFD(*p*, 0) ТОЧНОЕ(*p*,0)	␣␣␣␣␣␣␣␣$sd_1 d_2 \ldots d_m$, где *k* пробелов $m = p - (k-2)$; при $x \neq 0$ всегда $d_1 \neq 0$ (из этого условия определяется число *k*); при $x = 0$ $m = 1$	$p + 3$
FIXED(*p, q*) ТОЧНОЕ(*p, q*) $p > q > 0$	␣␣␣␣␣␣␣␣$sd_1 d_2 \ldots d_m, d_{m+1} \ldots d_{m+q}$ k пробелов где $m = (p - q) - (k - 1)$; при $x \neq 0$ всегда $d_1 \neq 0$ (из этого условия определяется число *k*); при $x = 0$ $m = 1$	$p + 3$
FIXED(*p*, q) ТОЧНОЕ(*p, q*) $q > 0$ и $p = q$	$s0.d_1 d_2 \ldots d_p$	$p + 3$

Обозначения: x — значение преобразуемого числа; d_i — десятичные цифры; s — либо минус, либо пробел; s' — либо минус, либо пробел;

Преобразование комплексных чисел в строки символов осуществляется в несколько этапов:

– коэффициенты при действительной и мнимой частях числа преобразуются в две строки символов по правилам, приведенным в табл. 5.1;

– правый пробел строки, соответствующий неотрицательному коэффициенту при мнимой части, заменяется знаком плюс;

– к этой же строке добавляется справа символ I;

– строки сцепляются с одновременным переносом всех пробелов, предшествующих второй строке (мнимая часть) в начало объединенной строки.

Длина строки символов, полученной в результате преобразования комплексного числа всегда равна $2l + 1$, где l — длина строки, получаемой при преобразовании любого из коэффициентов исходного числа (см. табл. 5.1). Двоичные числа предварительно преобразуются в десятичные, а лишь затем в строки символов. Из правил преобразования чисел в строки символов на практике чаще всего требуется знание длины строки, получаемой непосредственно после преобразования. Если, например, допустимая длина значений символьно-строчной переменной окажется меньше длины результата преобразования, то при присвоении числа символьно-строчной переменной будет потеряна часть правых символов константы, изображающей число.

Одним из частых случаев, когда программист имеет дело со строками символов, является печать различных сообщений и комментариев к результатам вычислений. Печать символьно-строчных значений может, в частности, осуществляться оператором вывода списка данных наравне с числовыми значениями. Обычно символьные строки печатаются не в виде констант языка PL/l, а в виде последовательности символов, представляющей значение строки (в гл. 10 будет уточнено, в каких случаях печатается константа и в каких — сама строка символов). Например, после выполнения оператора put list ('c''езд'); будет напечатано «c'езд».

В качестве простейшего примера использования символьно-строчных данных рассмотрим программу, анализирующую дискриминант квадратного уравнения, и печатающую на основе анализа одно из следующих сообщений: «корни комплексные», «корни действительные», «различные», «корни одинаковые», и кроме того, печатающую значения корней в виде комплексных чисел:

```
sqe: proc main;
dcl t char(32) var;
dcl (x1, x2, dc) float cplx;
dcl (a,b,b1,c,d) float;
get list(a, b, c);
d = b*b - 4*a*c;
if d < 0 then
t= 'корни комплексные'; else
if d > 0 then
т = 'корни действительные ,
различные';          else
т='корни_одинаковые';
dc = sqrt(complex(d, 0))/(2*a);
b1 = - b/(2*a);
x1 = b1+dc;
x2 = b1- dc;
put list(t, x1, x2);
end;
```

```
кв_корни: проц главная;
опс t текст(32) рд;
опс (x1, x2, dc) вещ комплексное;
опс (a,b,b1,c,d) вещ;
читать в_виде(a, b, c);
d = b*b - 4*a*c;
если d < 0 тогда
t= 'корни комплексные'; иначе
если d > 0 тогда
т = 'корни действительные ,
различные';          иначе
т='корни_одинаковые';
dc = sqrt(complex(d, 0))/(2*a);
b1 = - b/(2*a);
x1 = b1+dc;
x2 = b1- dc;
писать в_виде(t, x1, x2);
конец кВ_корни;
```

При выводе значений переменных символьно-строчного типа посредством оператора вывода именованных данных эти значения всегда представляются в форме соответствующих констант языка PL/1.

П р о в е р ь с е б я

1. Какие данные в языке PL/1 называются строками символов?
2. Каковы значения следующих констант

   1) 'РАЗ''ЕЗД'; 2) 'ХИ'(3); 3) ''
3. Что называется длиной строки символов?
4. Какие описатели типа значения могут быть приписаны символьно-строчным переменным?
5. Укажите область значений переменных, описанных оператором

   dcl (a, b var) char (80); опс (a, b рд) текст (80);
6. Пусть описан массив

   dcl м (100) char (3) var; опс м (100) текст (3) рд;

Могут ли различные элементы этого массива в конкретный момент иметь значения разной длины?

7. Пусть описаны переменные

dcl (a, b var) char (3), (c, d var) char (5);	опс (a, b рд) текст (3), (c, d рд) текст (5);

Каковы будут значения этих переменных после выполнения оператора

A, B, C, D='1234';

8. Пусть описаны переменные

dcl (a, b) float cplx, c fixed, e fixed (15, 5), f fixed (4, 5);	опс (a, b) вещ комплексное, c точное, e точное(15, 5), f точное(4, 5);

Какова будет длина каждой строки, полученной при автоматическом преобразовании значений указанных переменных к символьно-строчному типу?

9. Какие константы (десятичные или двоичные) представляются строкой, полученной при преобразовании двоичных чисел к символьно-строчному типу?

10. Пусть описана переменная

dcl a char (3);	опс a текст (3);

Каково будет ее значение после выполнения оператора

A='01'B;

11. Можно ли во входном потоке задать значение для символьно-строчной константы в виде, 'X'(80) ?

12. Напишите операторы, с помощью которых выводятся сообщения «ЕСТЬ НУЛИ» или «НЕТ НУЛЕЙ» ровно одним оператором вывода.

5.4. Операции над строками символов, встроенные функции для обработки строк

Специфической операцией, применимой к строкам символов, является *операция сцепления* (знак ||). Результатом операции является строка, полученная присоединением строкового значения 2-го операнда справа к строковому значению 1-го операнда. Например, пусть описаны переменные

dcl (a, b var) char (3), r char (9);	опс (a, b рд) текст (3), r текст (9);

Тогда после выполнения операторов

A, B='XY'; R=B || A || B; значение R будет равно 'XYXY␣XY␣␣'.

Напомним, что операция сцепления применима и к битовым строкам. В общем случае каждый операнд этой операции может быть произвольного типа: числового, битового, символьного. Однако операция непосредственно применима только к двум однотипным строкам. Если это условие не выполнено, что на практике встречается довольно редко, то автоматически производятся преобразования одного или обоих операндов по следующим

правилам. Если хотя бы один операнд является строкой символов или десятичным числом, то каждый операнд, отличный от строки символов, преобразуется в таковую. В противном случае двоичные числовые операнды операции сцепления преобразуются в строку битов. Правила преобразования чисел и строк битов в строку символов были рассмотрены в предыдущем параграфе, а правила преобразования двоичных чисел в битовые строки — в § 5.1.

К строкам символов применимы все операции сравнения. При этом если сравниваются строки различной длины, то меньшая строка считается дополненной справа необходимым количеством пробелов. Результат сравнения определяется на основе попарного сравнения соответствующих символов обеих строк. Причем строка $b_1b_2...b_n$ считается больше, чем строка $c_1c_2...c_n$ в том случае, если для некоторого $k \le n$ выполняются условия

$$b_1 = c_1, \quad b_2=c_2, \ \dots \ b_{k-1} = c_{k-1} \quad b_k > c_k$$

Отдельный символ b считается большим, чем символ c, если битовая строка, являющаяся внутренним представлением b, больше битовой строки, являющейся внутренним представлением c. Внутренние представления символов будут рассмотрены в § 13.5, здесь же ограничимся перечислением в порядке возрастания символов:

Меньший код внутреннего представления имеет пробел, затем идут скобки и знаки, цифры от 0 до 9, прописные латинские, строчные латинские, прописные русские, строчные русские. Буквы Ё и ё имеют отдельное место, а не между кодами Е и Ж.

Рассмотрим примеры. Пусть значение символьно-строчной переменной A равно 'ACE␣'. Тогда следующие логические выражения истинны:

A='ACE'; A='ACE␣␣'; A>'ACD'; A<'ACH'; A<'ACE.'

Если один из операндов операции сравнения — строка символов, а другой — строка битов, то последний автоматически преобразуется в строку символов по рассмотренным выше правилам.

Если один из операндов операции сравнения — строка символов, а другой — число, то строка преобразуется в число. При этом строка символов должна изображать любую допустимую в языке PL/1 константу или комплексное выражение, допустимое для входного потока оператора GET/ЧИТАТЬ (см. § 2.4). В начале и конце строки может находиться произвольное число пробелов. Пустая строка допустима, она преобразуется в число ноль. Отметим, что если числовой операнд операции сравнения представлен в форме с фиксированной точкой, то строковый операнд всегда преобразуется в целое число.

Выражения с символьно-строчными значениями могут быть использованы в качестве операндов арифметических и логических операций (что, однако, редко используется на практике). В первом случае строка символов преобразуется в число так же, как и при операциях сравнения (см. выше). Во втором случае строка символов преобразуется в строку битов; при этом символ 0 преобразуется в бит 0, а символ 1 в бит 1 (другие символы в исходной строке недопустимы).

Символьно-строчные значения можно присваивать переменным с числовыми или битовыми значениями. При этом строка символов автоматически преобразуется в соответствующее число или битовую строку по правилам, рассмотренным выше для операндов операций. Однако в данном случае (в отличие от преобразования символьно-строчных операндов операций сравнения с числами или арифметических операций) при преобразовании строки символов в число с фиксированной точкой дробная часть числа не теряется. Например, если описаны переменные

```
dcl (a, b var) char (5), c bit (3), (x, y)
fixed (5, 2);
```

```
опс (a, b рд) текст(5), с бит(3), (x, y)
точное(5, 2);
```

то после выполнения операторов

```
A='␣1.2␣';  B='101';  X=A;  C, Y=B;
```

значение X будет равно 1.20, значение Y будет равно 101.00, а значение C='101'B.

Из встроенных функций при работе со строками символов наиболее часто используется функция **SUBSTR/ПОДСТРОКА** и псевдопеременная с тем же именем. Как и в случае битовых строк, здесь функция SUBSTR/ПОДСТРОКА применяется для обращения к подстроке некоторой строки символов, а псевдопеременная SUBSTR/ПОДСТРОКА позволяет изменять определенную подстроку текущего значения символьно-строчной переменной. Обращение и к функции, и к псевдопеременной имеет в данном случае вид SUBSTR(*e, k, l*) или ПОДСТРОКА(*e, k, l*) или SUBSTR(*e, k*) или ПОДСТРОКА(*e, k*), где *e* для функции — выражение с символьно-строчным значением, а для псевдопеременной — имя переменной символьно-строчного типа, значение которой изменяется. Аргумент *k* задает номер символа, с которого начинается подстрока, а аргумент *l* — длину подстроки (если он опущен, то подстрока включает весь остаток строки, начиная с *k*-го символа). Например, если значение символьно-строчной переменной A равно 'ABCD', то значения указателей функций SUBSTR(A, 2, 2)/ПОДСТРОКА(A, 2, 2) и SUBSTR (A, 3)/ПОДСТРОКА(A, 3) будут, соответственно, равны 'BC' и 'CD'.

Другой пример: пусть имеется описание

dcl (a, b var) char (5); опс (a, b рд) текст(5);

и значения переменных A и B, соответственно, равны 'тонна' и 'метр'. Тогда после выполнения операторов

substr (a, 3,2) ='пк'; подстрока (a, 3,2) ='пк';
substr (b, 3) ='ра'; подстрока (b, 3) ='ра';

значения переменных станут равны, соответственно, 'топка' и 'мера'. Отметим, что с помощью псевдопеременной SUBSTR/ПОДСТРОКА нельзя изменить длину текущего значения переменной. Например, если значение описанной выше переменной B равно 'метр', то любой из следующих операторов присваивания неверен:

1) substr (b, 5, 1)='ы'; подстрока (b, 5, 1)='ы';
2) substr (b, 4, 2)='ры'; подстрока (b, 4, 2)='ры';
3) substr (b, 5)='ы'; подстрока (b, 5)='ы';

Строка, присваиваемая псевдопеременной SUBSTR/ПОДСТРОКА, может быть и короче, и длиннее подстроки. В этом случае присваиваемое значение усекается справа или дополняется справа пробелами до длины подстроки. Например, если значение переменной A равно строке 'паровоз', то после выполнения операторов

substr (a, 3, 2)= 'н'; подстрока (a, 3, 2)= 'н';
substr (a, 7) = 'ра'; подстрока (a, 7) = 'ра';

ее значение станет равно строке 'пан␣вор'.

Для анализа строк символов удобно использовать встроенные функции **INDEX/ИСКАТЬ** и **VERIFY/ИСКАТЬ_НЕСОВПАДЕНИЕ**. Значением функции INDEX(c_1, c_2, n) или ИСКАТЬ(c_1, c_2, n), где c_1 и c_2 — выражения с символьно-строчными значениями c'_1 и c'_2 является номер левого символа строки c'_1, начиная с которого строка c'_2 целиком входиг в строку c'_1. Если же строка c'_2 вообще не входит в c'_1, то значением функции INDEX/ИСКАТЬ будет число ноль. Целая константа n задает номер символа, начиная с которого нужно искать совпадение. Если n не задан, он считается равным 1. Если $n < 0$, то поиск в c'_1 начнется справа, т.е. с конца строки.

Например, если значение переменной A равно 'слово', то значениями указателей функций INDEX(A, 'лов'), ИСКАТЬ(A, 'лов'), и INDEX(A, '␣') ИСКАТЬ(A, '␣') будут, соответственно, 2 и 0. Значение функции INDEX/ИСКАТЬ является вещественным двоичным числом в форме с фиксированной точкой с точностью (15).

Значением функции VERIFY/ИСКАТЬ_НЕСОВПАДЕНИЕ (c_1, c_2), где c_1 и c_2 — выражения с символьно-строчными значениями c'_1 и c'_2 является номер

левого символа строки c'_1 , который не входит в строку c'_2. Если же все символы строки c'_1 входят в строку c'_2. (в любом порядке), то значение функции будет равно нулю. Пусть, например, значением символьно-строчной переменной B является последовательность всех букв русского алфавита, а значением переменной A является строка 'пробный␣␣текст'. Тогда значения указателей функций VERIFY(A, B), VERIFY(SUBSTR (A, 8), '␣') и VERIFY (SUBSTR(A, 7), B) равны, соответственно, 8, 3 и нулю. Значение функции VERIFY является вещественным двоичным числом в форме с фиксированной точкой с точностью (15).

Для замены во всей строке одних символов на другие удобно использовать встроенную функцию **TRANSLATE/ПЕРЕВЕСТИ**. Значением функции TRANSLATE(c_1, c_2, c_3) или ПЕРЕВЕСТИ(c_1, c_2, c_3), где c_1, c_2 и c_3 — выражения с символьно-строчными значениями c'_1, c'_2, c'_3, является строка символов такой же длины, что и c'_1. Если i-й символ c'_1 не входит в строку c'_3, то он будет равен i-му символу строки-результата. Иначе пусть i-й символ строки c'_1 равен j-му символу строки c'_3. Тогда i-й символ строки-результата будет равен j-му символу строки c'_2. Например, значением указателя функции TRANSLATE('ЗА␣4Т0', 'ЗЧО', '340') будет строка 'ЗА␣ЧТО'. Для замены большого числа различных символов лучше использовать другой вариант обращения к функции TRANSLATE/ПЕРЕВЕСТИ, а именно TRANSLATE(c_1 c_2). При этом 3-й аргумент функции считается равным строке из 256 символов, в которой внутреннее представление i-го символа совпадает с представлением числа (i - 1) в виде восьмизначного двоичного числа.

Значением встроенной функции **CHAR(*e*, *k*)**, или **ТЕКСТ(*e*, *k*)**, где e — выражение со значением любого типа, k — целая десятичная константа, является результат преобразования значения e к строке символов, дополненный справа пробелами или усеченный справа до длины k символов. При преобразовании применяются стандартные правила (см. § 5.3). Функция CHAR/ТЕКСТ применима и при работе только со строками символов. Например, если описана переменная

```
dcl a char (80) var;                    опс  а текст (80) рд;
```

и требуется присвоить ей строку, состоящую из символа 'x', за которым следует 20 пробелов, то это можно сделать с помощью оператора

```
a=char('x', 21);                        а=текст('x', 21);
```

Заметим, что в рассматриваемой версии языка введена еще одна форма этой функции, когда длина строки указывается со знаком «минус». В этом случае символы усекаются не справа, а слева.

Значение встроенной функции **LENGTH(*e*)** или **ДЛИНА(*e*)**, где *e* — скалярное выражение со значением символьно-строчного типа, равно длине (в символах) значения *e*. Пример использования встроенной функции LENGTH/ДЛИНА с аргументом символьно-строчного типа рассмотрен в § 5.5.

Для иллюстрации большинства изложенных возможностей манипулирования строками символов приведем программу, которая вводит текст на русском языке, анализирует его и определяет среднюю длину слова и частоту появления каждой из букв алфавита в конце слов (в пересчете на 100 слов). Условимся, что во входном потоке текст представлен в виде совокупности строк, заключенных в апострофы, длина каждой строки не более 254 символов, и что текст не содержит символов, отличных от русских букв, пробелов, точек, запятых, точек с запятыми и двоеточий. Программу будем рассматривать по фрагментам.

Вначале введем текст, а затем для упрощения выделения слов заменим в нем все разделители на пробелы и добавим в конце пробел:

```
анализ_текста: процедура главная;
```

подготовим все необходимое для анализа текста:

```
опс t          текст(254) рд;
опс а(33)      текст;              /* алфавит */
опс n(33)      точное(31);         /* счетчики букв */
опс nw         точное(31);         /* счетчик слов */
опс nl         точное(31);         /* общий счетчик букв */
опс м          точное(31);         /* текущая позиция в тексте */
а = 'абвгдеёжзийклмнопрстуфхцчшщьыъэюя';

когда конец_файла(стд_ввод) идти ввод_окончен;
читать_строку:
читать в_виде(t);
n = 0;
nw =0;
nl = 0;
м= 1;
//---- переводим все знаки в пробелы и добавляем пробел в конце ----
t= перевести(t, '␣' (4), '.,;:')|| '␣';
//---- переводим все строчные буквы в прописные ----
t= перевести(t, 'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЬЫЪЭЮЯ',
                'абвгдеёжзийклмнопрстуфхцчшщьыъэюя');
```

Далее следует анализ текста, состоящий из этапов:

– поиск начала очередного слова (пропуск пробелов);
– поиск конца слова (поиск пробела);
– определение номера последней буквы слова:

```
анализ: j = искать_несовпадение(подстрока(t, м), '␣');
если j = 0 тогда идти читать_строку;  /*слова кончились*/
nw+=1;
m +=  j - 1;
k = искать(подстрока(t, m), '␣');
nl += k - 1;
m += k - 1;
цикл i = 1 до 33;
        если подстрока(t, m - 1, 1) = подстрока(a, i, 1) тогда
        {;
           n(i) +=1;
           идти анализ;
         };
конец i;
идти анализ;
```

Отметим, что вместо оператора

```
k = искать(подстрока(t, m), '␣');
```

для поиска конца слова можно было бы использовать оператор

```
k = искать_несовпадение (подстрока (t,m), a);
```

в котором ищется символ, отличный от буквы. Если бы мы не заменили разделители на пробелы, то пришлось бы использовать оператор подобного вида.

В заключении выведем полученные результаты. Причем сведения о частоте появления букв в конце слов будем печатать в виде частота буква

```
ввод_окончен: писать в_виде('средняя длина слова =',nl/nw, 100*n/nw||a);
конец анализ_текста /*конец программы*/;
```

Обратите внимание на использование в качестве операндов операции сцепления чисел, а также массивов.

Имеются еще встроенные функции для работы с символьно-строчными переменными. Их можно считать надстройкой над уже описанными выше встроенными функциями. Встроенная функция **TRIM/ОЧИСТИТЬ*(c)***, где c выражение с символьно-строчным значением возвращает строку c' с убранными справа и слева символами пробела. Обратите внимание, что

переменная, куда записывается результат работы этой встроенной функции, должна иметь переменную длину, иначе пробелы могут быть дописаны вновь.

Встроенная функция **REPLACE/ЗАМЕНИТЬ(c_1, c_2, c_3)** во многом подобна функции TRANSLATE/ПЕРЕВЕСТИ, однако меняет в результирующей строки не отдельные символы из c_2 , найденные в c_3, а целиком фрагмент c_3 на фрагмент c_2. Например, оператор

s=ЗАМЕНИТЬ(s, 'палки' , 'елки');

сформирует из символьно-строчной переменной s строку, где все слова «елки» будут заменены на «палки». При этом (в отличие от TPANSLATE/ПЕРЕВЕСТИ) длина результирующей строки может измениться. Если вхождений образца c_3 в исходную строку c_1 не найдено, результирующая строка совпадет со строкой c_1. Так же, как и в случае функции TRIM/ОЧИСТИТЬ, переменная, куда присваивается результирующая строка, должна иметь переменную длину, иначе результат может быть обрезан или дополнен пробелами.

Встроенная функция **TALLY/ПОДСЧЕТ(c_1, c_2)** выдает результат применения функции INDEX/ИСКАТЬ(c'_1, c_2) ко всем символам символьно-строчного выражения c_1, точнее, если применение INDEX/ИСКАТЬ к подстроке с очередной позиции дает единицу, эта единица учитывается в общем результате. Таким образом, TALLY/ПОДСЧЕТ подсчитывает, сколько раз заданный образец встретился в заданной строке, например

ПОДСЧЕТ('xxxx','xx') возвращает 3

TALLY ('0x123456789x','x') возвращает 2

П р о в е р ь с е б я

1. Пусть описаны переменные

   dcl(a, b var) char (4), r char (12); опс(a, b рд) текст (4), r текст (12);

   каково будет значение переменной R после выполнения операторов

   A, B='ABC'; R = B|| A || B;

2. Могут ли оба операнда операции сцепления быть числами? Как в этом случае выполняется операция?

3. Пусть значение переменной A равно 'все␣'. Укажите, какие из следующих логических выражений истинны:

1) A='все'; 2) A^='все␣␣'; 3) A > 'все.'; 4) A<'всд';.

4. Какой из операндов операции сравнения преобразуется, если один из операндов — строка символов, а другой — число?

5. Пусть описана переменная опс В бит (1); каково будет ее значение после выполнения оператора В='1';

6. Пусть значение переменной А равно 'ABABC'. Вычислите значения следующих указателей функций:

1) ПОДСТРОКА(А, 2, 3); 2) ПОДСТРОКА(А, 3);
3) ИСКАТЬ(А, 'В'); 4) ИСКАТЬ(А, 'D');
5) ИСКАТЬ_НЕСОВПАДЕНИЕ(А, 'ВА');
6) ИСКАТЬ_НЕСОВПАДЕНИЕ(А, 'АВС');
7) ПЕРЕВЕСТИ, 'XY', 'AB'); 8) ТЕКСТ (А, 10).

7. Пусть значение переменной А равно строке 'разговор'. Каково станет ее значение после выполнения всей последовательности операторов

ПОДСТРОКА (А, 1,2) ='он';
ПОДСТРОКА (А, 5, 2) = 'и';
ПОДСТРОКА (А, 8) = 'н';
ПОДСТРОКА (А, 3,2) ='а';

8. Может ли измениться длина результирующей строки относительно исходной при вызове встроенной функции REPLACE/ЗАМЕНИТЬ?

9. Переменная А постоянной длины описана как опс А текст(3) и содержит значение '1␣␣'; Изменится ли она после оператора А=ОЧИСТИТЬ(А);

10. Как необходимо изменить рассмотренную в данном параграфе программу анализ_текста для того, чтобы она могла обрабатывать тексты, дополнительно содержащие восклицательные и вопросительные знаки?

11. Как необходимо изменить рассмотренную в данном параграфе программу анализ_текста для того, чтобы она обрабатывала тексты на английском языке, а не на русском (с теми же разделителями)?

12. Составьте программу, которая вводила бы одну строку символов — текст, состоящий из целых чисел (возможно со знаком), разделенных пробелами; в результате работы программы должны быть напечатаны среднее количество цифр в числе и частота появления каждой из цифр в начале числа (в пересчете на 100 чисел).

5.5. Параметры и функции со значениями строкового типа

В процедурах допускаются параметры, в том числе и параметры-массивы, со значениями строкового типа. Такие параметры должны быть описаны внутри процедуры в операторе DECLARE/ОПИСАНИЕ вместе с описателем CHARACTER/ТЕКСТ или BIT/БИТ и, возможно, с описателем длины и

описателем VARYING/РАЗНОЙ_ДЛИНЫ. Аргумент, сопоставляемый параметру символьно-строчного типа, должен быть выражением со значением типа строки символов, аргумент, сопоставляемый параметру битово-строчного типа, должен быть выражением со значением типа строки битов. Для параметров-массивов аргумент должен быть массивом.

Если параметру строкового типа приписан описатель VARYING/РАЗНОЙ_ДЛИНЫ или VAR/РД, то аргумент, сопоставляемый этому параметру, должен иметь значение изменяющейся длины, и наоборот, если параметру не приписан описатель VARYING/РАЗНОЙ_ДЛИНЫ, то соответствующий ему аргумент должен иметь значение постоянной длины. Для того чтобы определить, имеет ли значение выражения постоянную или изменяющуюся длину, необходимо использовать следующие правила. Операнд выражения, являющийся именем переменной (массива) или указателем функции (не встроенной), имеет значение изменяющейся длины тогда и только тогда, когда этой переменной (массиву) или функции приписан описатель VARYING/РАЗНОЙ_ДЛИНЫ. Операнд, являющийся константой, а также операнд, значение которого равно результату преобразования числа в строку, имеет значение постоянной длины. Результат операции сравнения тоже имеет всегда постоянную длину. Результат операции сцепления или логической операции имеет изменяющуюся длину, если хотя бы у одного операнда значение изменяющейся длины.

Описатель длины для параметров строкового типа может иметь вид

$$(*) \text{ или } (n)$$

где n — целая десятичная константа. Этот описатель может быть опущен, при этом по умолчанию предполагается описатель (1). Если в описателе длины параметра указана константа, то соответствующий аргумент должен иметь значение именно с такой длиной (максимальной в случае значений изменяющейся длины). Если же в описателе длины параметра указана звездочка, то длина значения каждого сопоставленного ему аргумента может быть произвольной (для значений изменяющейся длины произвольной может быть их максимальная длина).

Фиктивные аргументы сопоставляются параметрам строкового типа в тех же случаях, что и для параметров других типов: во-первых, если аргумент отличен от имени переменной (напомним, что для имени переменной, заключенного в скобки, всегда создается фиктивный аргумент); во-вторых, если тип значения аргумента не отвечает рассмотренным выше требованиям к соответствию типу параметра и при этом тип параметра описан в описателе

ENTRY/ДЛЯ_ВЫЗОВА в вызывающей процедуре либо параметр принадлежит внутренней процедуре.

Обычно тип параметра описывается в описателе ENTRY/ДЛЯ_ВЫЗОВА в вызывающей процедуре в тех случаях, когда необходимо обеспечить автоматическое преобразование типа значения аргумента к типу соответствующего параметра (например, если параметру строкового типа сопоставляется аргумент с числовым значением).

Параметр строкового типа описывается внутри описателя ENTRY/ДЛЯ_ВЫЗОВА с помощью тех же описателей (CHARACTER/ТЕКСТ, BIT/БИТ, VARYING/РАЗНОЙ_ДЛИНЫ, описатель длины), что и в операторе DECLARE/ОПИСАНИЕ. Напомним, что в языке PL/1 описатель ENTRY/ДЛЯ_ВЫЗОВА должен быть задан для всех имен внешних процедур.

Пример. Хотя есть встроенная функция TRIM/ОЧИСТИТЬ, рассмотрим свою процедуру, которая для любой переменной с символьно-строчными значениями изменяющейся длины отбрасывает в текущем значении переменной все находящиеся справа пробелы:

```
xpr: pr oc (c);
dcl c char(*) var, i fixed;
do i = length(c) by - 1 to 1
while(substr(c, i, 1) = '␣');
end;
c = substr(c, 1,i);
end;
```

```
xpr: проц (с);
опс с текст(*) рд, i точное;
цикл i = длина(с) с_шагом - 1 до 1
пока(подстрока(с, i, 1) = '␣');
конец;
с = подстрока(с, 1,i);
конец;
```

Так как задачей данной процедуры является изменение значения аргумента, то, следовательно, при ее вызове недопустимо создание фиктивных аргументов. Поэтому аргумент, указанный в операторе вызова процедуры XPR, всегда должен быть именем переменной с символьно-строчными значениями изменяющейся длины.

Процедуры-функции могут вычислять значения типа строки битов или символов. Оператор PROCEDURE/ПРОЦЕДУРА или ENTRY/ДЛЯ_ВЫЗОВА, задающий подобный вход в процедуру, должен содержать конструкцию RETURNS/ВОЗВРАЩАЕТ (см. § 4.2) с описанием вычисляемого значения. При этом используют обычные описатели строковых значений: CHARACTER/ТЕКСТ, BIT/БИТ, VARYING/РАЗНОЙ_ДЛИНЫ и описатель длины. В языке PL/1 в описателе длины внутри конструкции RETURNS/ВОЗВРАЩАЕТ может быть задана или константа, или звездочка.

Рассмотрим примеры. Следующая процедура-функция с одним параметром — одномерным числовым массивом, имеет значение '1'B, если

значения всех элементов массива-параметра равны между собой; в противном случае значение функции равно '0'B:

```
eqa: proc(a) returns(bit(1));
dcl a(1000) float, i fixed;
do i = lbound(a) to hbound (a) - 1;
    if a(i) ^= a(i+1) then return('0'b);
end;
return('1'b);
end;
```

```
eqa: проц(a) возвращает(бит(1));
опс a(1000) вещ, i точное;
цикл i = ниж_граница(a) до
верх_граница (a) - 1;
  если a(i) ^= a(i+1) тогда возврат('0'б);
конец;
возврат('1'б); конец;
```

В вызывающей процедуре потребуется описание вида

```
dcl eqa returns (bit (i));
```

```
опс eqa возвращает(бит(1));
```

Другой пример. Следующая процедура-функция с одним числовым параметром имеет символьно-строчное значение изменяющейся длины, изображающее константу, равную исходному числу, причем константа не содержит нулей в конце дробной части, а для целых чисел она не содержит и десятичной точки (подобные процедуры могут применяться при редактировании выводимых числовых данных)

```
fconst: proc(a) returns(char (13) var);
dcl a fixed(10, 5), c char (13), i
fixed;
c = a;
do i = 13 by -1 to 9
while(substr(c, i, 1) = '0';
end;
if i = 8 then i=i - 1;
/*в 8-й позиции расположена
точка */
return(substr(c, 1,i));
end;
```

```
fconst: проц(a) возвращает(текст(13)
рд);
опс a точное(10, 5), c текст(13), i
точное;
c = a;
цикл i = 13 с_шагом -1 до 9
пока(подстрока(c, i, 1) = '0';
конец;
если i = 8 тогда i=i - 1;
/*в 8-й позиции расположена точка */
возврат(подстрока(c, 1,i));
конец;
```

В вызывающей процедуре требуется описание вида

```
dcl fconst entry(fixed(10,5)) returns(char(13) var);
```

или

```
опс fconst для_вызова(точное(10,5)) возвращает(текст(13) рд);
```

П р о в е р ь с е б я

1. Какие описатели применяются при описании значений параметров строкового типа?

2. Каковы требования к соответствию между аргументами и параметрами строкового типа?

3. В каких случаях результат операции сцепления имеет значение изменяющейся длины?

4. Можно ли в описателе длины параметров (в операторе описания) задавать выражения, отличные от константы?

5. Составьте процедуру-функцию, вычисляющую количество пробелов в значении параметра символьно-строчного типа; составьте оператор описания, описывающий имя этой процедуры с указанием типа параметра.

6. Составьте процедуру, которая отбрасывает находящиеся справа нулевые биты в текущем значении любой переменной с битово-строчными значениями изменяющейся длины.

7. Какую конструкцию всегда должен содержать оператор, задающий вход в процедуру-функцию со значением строкового типа?

8. Может ли быть указана звездочка в описателе длины функции внутри описателя RETURNS?

9. Составьте процедуру-функцию с одним параметром — одномерным числовым массивом, которая имеет значение '1'b, если в массиве есть хотя бы два смежных элемента с равными значениями; в противном случае значение функции должно быть равно '0'b. Составьте оператор описывающий имя этой процедуры с указанием типа вычисляемого значения.

10. В процедуру функцию fconst, рассмотренную в конце параграфа, внесите изменения таким образом, чтобы она решала ту же задачу, но для параметров со значениями с точностью (15, 10). Составьте оператор описывающий имя модифицированного варианта процедуры fconst с указанием типа вычисляемого значения.

6. РЕДАКТИРОВАНИЕ ДАННЫХ ПРИ ВВОДЕ И ВЫВОДЕ

6.1. Операторы ввода и вывода с редактированием

Под редактированием данных при вводе или выводе понимается, во-первых, управление формой представления вводимых или выводимых данных и, во-вторых, управление расположением данных во входном или выходном потоке. Рассмотренные ранее варианты операторов ввода (GET/ЧИТАТЬ) и вывода (PUT/ПИСАТЬ) не позволяют управлять формой представления данных и допускают лишь ограниченные возможности управления размещением данных во входном или выходном потоке. Наиболее полно возможности редактирования данных реализуются операторами ввода или вывода с ключевым словом **EDIT/В_ФОРМЕ**, имеющими в простейшем случае вид

1) **GET EDIT** $s_1 f_1\ s_2 f_2\ \ldots s_n f_n$ или **ПИСАТЬ В_ФОРМЕ** $s_1 f_1\ s_2 f_2\ \ldots s_n f_n$
2) **PUT EDIT** $s_1 f_1\ s_2 f_2\ \ldots s_n f_n$ или **ЧИТАТЬ В_ФОРМЕ** $s_1 f_1\ s_2 f_2\ \ldots s_n f_n$

Здесь s_i $(i = 1, 2, \ldots, n)$ — список данных, синтаксически и семантически аналогичный спискам данных операторов GET/ЧИТАТЬ или PUT/ПИСАТЬ с ключевым словом LIST/В_ВИДЕ, f_i — список форматов, содержащий сведения, необходимые для редактирования данных. Эти сведения задаются элементами формата двух типов: элементами формата данных и управляющими элементами формата.

Список форматов f_i имеет в общем случае вид

$$(k_1 e_1,\ k_2 e_2,\ \ldots,\ k_n e_n)\ \ (n \geq 1).$$

где k_j $(j=1, 2, \ldots, n)$ — либо пусто, либо повторитель формата, являющийся целой десятичной константой; e_j,-— либо произвольный элемент формата, либо заключенный в скобки подсписок форматов, который в свою очередь может включать как отдельные элементы форматов, так и внутренние подсписки форматов. Перед любым подсписком должен быть задан повторитель формата.

Каждому элементу данных, вводимому или выводимому в соответствии со списком данных s_i, сопоставляется один элемент формата из списка f_i. Именно этот элемент формата определяет представление вводимого или выводимого значения элемента данных. Сопоставление элементу данных элемента формата осуществляется по следующим правилам. Для первого элемента данных списка s_i, поиск необходимого элемента формата производится слева направо от начала списка f_i. Для каждого последующего

элемента данных список продолжается вправо, начиная от последнего использованного элемента формата данных. При достижении конца списка форматов повторяется поиск от начала списка f_i. Например, в операторе

ЧИТАТЬ В_ФОРМЕ(A, B, C) (x, y) (D, E) (z),

где *A, B, C, D* и *E* — имена переменных, *x*, *y* и *z* — элементы формата данных, значение переменной *A* будет вводиться в соответствии с элементом формата *x*, значение переменной *B* — в соответствии с *y*, значение переменной *C* — в соответствии опять с *x*, значение переменных *D* и *E* — в соответствии с *z*.

Если в процессе просмотра списка форматов встречается элемент формата с повторителем, то далее этот элемент применяется k раз, где k — текущее значение повторителя. Если же встречается подсписок форматов, то далее этот подсписок просматривается от начала до конца k раз, где k — текущее значение повторителя перед списком. Например, при выполнении оператора

ЧИТАТЬ В_ФОРМЕ(A, B, C, D, E, F, G, H, I, J, K) ((2) (2) (y, 3 z));

где *A, B, C, D, E, F, G, H, I, J, K* — имена переменных, *x*, *y* и *z* — элементы формата данных, будет установлено следующее соответствие между значениями переменных и элементами формата:

A~x, B~x, C~y, D~z, E~z, F~z, G~y, H~z, I~z, J~z, K~x.

Имеется 6 элементов формата данных, они обозначаются буквами **A, B, C, E, F** и **P**. У некоторых элементов имеются и обозначения кириллицей, причем такие обозначения допустимы только внутри списка форматов. В остальной программе так можно обозначать обычные переменные.

A эквивалентно **Т** (текст)
B эквивалентно **Б** (биты)
F эквивалентно **Ч** (числа)
P эквивалентно **Ш** (шаблон)

Элемент формата **A** или **A(*n*)**, где n — целочисленная константа, при выводе указывает, что значение выводимого элемента данных должно быть преобразовано в строку символов по стандартным правилам (см. § 5.3); если задана константа n, то результат преобразования дополняется справа пробелами или усекается справа до длины, равной значению n. Например, после выполнения оператора

ПИСАТЬ В_ФОРМЕ(':') (A) (1, 11) (A(11), A(15));

будет выведена последовательность символов «:␣␣␣␣␣␣␣␣␣1␣␣␣␣␣␣␣␣11»

При вводе элемент формата A(*n*) указывает, что вначале из очередных *n* символов входного потока формируется строка символов, которая затем преобразуется к типу значения соответствующей переменной по стандартным правилам (см. § 5.3, § 5.4). Например, пусть переменные A, B и C описаны оператором опс a точное, b бит(2), c текст (3);
и пусть входной поток имеет вид 1111111 ... Тогда после выполнения оператора ЧИТАТЬ В_ФОРМЕ(A, B, C) (A(2)); значение переменной *A* станет равным 11, значение переменной *B* станет равным '11'B, значение переменной *C* станет равным '11␣'. Русский эквивалент формата A – буква Т (текст).

Элемент формата **B** или **B(*n*)**, где *n* — целочисленная константа, при выводе указывает, что значение выводимого элемента данных вначале должно быть преобразовано в строку битов по стандартным правилам (см. § 5.1), затем преобразовано в строку символов и, если задана константа *n*, дополнено справа пробелами или усечено до длины, равной значению *n*. Например, после выполнения оператора

ПИСАТЬ В_ФОРМЕ('101'Б, '11'Б) (B(5), B);

будет выведена последовательность символов «␣␣010111».

При вводе элемент формата B(*n*) (здесь константа *n* всегда должна быть задана) указывает, что очередные *n* символов входного потока должны включать последовательность цифр 1 и 0, задающую значение битовой строки. Последовательность цифр 1 и 0 может быть окружена пробелами. Никакие другие символы, помимо двоичных цифр и пробелов, недопустимы; между цифрами пробелы также недопустимы. Введенная битовая строка затем преобразуется по стандартным правилам (см. § 5.2) к типу значения соответствующей переменной.

Например, пусть переменные D и E описаны оператором

опс D бит(4), E бит(3);

и пусть входной поток имеет вид 101101101... Тогда после выполнения оператора читать в_форме(D, E) (B(4), B(3));
значения переменных станут D= '1011'b и E= '011'b.

Имеются разновидности формата *B*: **B1*(n)*, B2*(n)*, B3*(n)*,** и **B4*(n)***, которые вводят и выводят битовые значения в разных системах счисления, а именно в двоичной, в четверичной, восьмеричной и шестнадцатеричной.

Обозначение формата B и B1 эквивалентны. Для формата B2 допустимы символы входного потока 0, 1, 2, 3. Для формата B3 допустимы символы 0, 1,

2, 3, 4, 5, 6 и 7. Для формата B4 допустимыми являются символы 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Русские эквиваленты обозначений этих форматов **Б1, Б2, Б3** и **Б4** соответственно. Например, если битово-строчная переменная A длиной 16 содержит все единицы, то ее вывод в разных форматах даст:

ПИСАТЬ В_ФОРМЕ(A)(Б1);	1111111111111111
ПИСАТЬ В_ФОРМЕ(A)(Б2);	33333333
ПИСАТЬ В_ФОРМЕ(A)(Б3);	177777
ПИСАТЬ В_ФОРМЕ(A)(Б4);	FFFF

Элемент формата данных **F** применяется при вводе и выводе вещественных чисел в форме с фиксированной точкой. Он может иметь вид F(*n*), или F(*n*, *q*), где n и q — целочисленные константы. Русский эквивалент этого формата **Ч** (от слова «число»);

При выводе формат F(*n*, *q*) указывает, что вначале значение выводимого элемента данных преобразуется в десятичное число в форме с фиксированной точкой с точностью (15, q). При этом применяется не стандартное преобразование, а преобразование с округлением: если отбрасываемая часть числа больше или равна по абсолютной величине $5 \cdot 10^{-(q+1)}$, то к результату преобразования прибавляется 10^{-q}, умноженное на знак числа. Затем полученное число преобразуется по стандартным правилам в строку символов, которая далее дополняется слева пробелами или усекается слева до длины, равной значению выражения n. Например, после выполнения оператора

```
писать в_форме(66,-5.5E-3) (F(2,0), F (8, 3));
```

будет выведена последовательность символов 66 ␣ ␣-0.006

Формат F(*n*) при выводе эквивалентен формату F(*n*, *0*).

При вводе формат F(*n*) указывает, что вводимое число должно быть представлено десятичной константой с фиксированной точкой, расположенной в очередных n символах входного потока; константа может быть окружена пробелами. Если все очередные n символов являются пробелами, то вводится число нуль. Например, если описан массив A:

```
dcl a(4) fixed (15, 5);
```

и выполнен оператор

```
get edit (a) (f (5));
```

то при наличии входного потока « ␣-15␣ - .002␣␣␣␣5.55 ␣»
значения элементов массива A станут, соответственно, равны -15, -0,002, 0 и 5,55.

Формат F(n,q) при вводе применяется, когда желательно опускать точку в константах, изображающих вводимое число. А именно, если в константе нет точки, то по умолчанию считается, что последние q цифр константы являются цифрами дробной части числа. Если же в константе точка есть, то ввод выполняется как для формата F(n). Например, если для описанного выше массива выполнен оператор

get edit (a) (f(5, 2));

и входной поток имеет вид « ␣-15␣ - .002␣␣ 1.4444 ␣␣ »
то значения элементов массива A станут, соответственно, равны -0,15, -0,002, 1,4 и 4,44.

Элемент формата данных **E** применяется при выводе вещественных чисел в форме с плавающей точкой и при вводе чисел в форме с плавающей и фиксированной точкой. Он может иметь вид E(n) или E(n, q), где n, q — целочисленные константы.

При выводе формат E(n, q) указывает, что выводимое число должно быть изображено десятичной константой с плавающей точкой всего n символов, мантисса которой содержит q цифр в дробной части. Причем порядок константы подбирается таким образом, чтобы первая цифра целой части для ненулевых чисел была бы не равна нулю. Полученная константа дополняется слева пробелами или усекается слева до длины, равной значению n.

Например, после выполнения оператора: PUT EDIT (33.2) (E(11, 3)); будет выведена последовательность символов ␣ 3.320E+01

При преобразовании в константу выводимое число округляется по таким же правилам, что и для формата F. Например, если переменной A со значениями в форме с плавающей точкой было присвоено число 13,1, то после выполнения оператора PUT EDIT (A) (F(12, 5)); будет выведена константа 1.31000E+01, а после выполнения оператора PUT LIST(A); будет выведена константа 1.310000E+01.

Формат E(n, q) при вводе указывает, что вводимое число должно быть представлено десятичной константой в форме с плавающей или с фиксированной точкой. Константа должна занимать очередные n символов входного потока и может быть окружена пробелами. В отличие от формата F в данном случае все n символов не могут быть пробелами. Если $q \neq 0$ и константа не содержит точки, то считается, что точка расположена слева от последних q цифр константы (не считая цифр порядка). Например, если описан массив A как ОПС A (6) ВЕЩ; и выполнен оператор

GET EDIT (A) (E (6, 0), E (6, 1)); то при наличии входного потока

␣ -120 ␣␣ -120 ␣ ␣12E-2 ␣ 12E -2 ␣ 1.2E5 1.2 ␣ ␣ ␣ ␣

элементам массива A будут присвоены значения: -120, -120, 0,12, 0,12, 120000, 1,2.

Элемент формата данных ***C*** применяется при вводе и выводе комплексных чисел. Он имеет вид C(*f*), где *f* — произвольный элемент формата данных. Формат C(*f*) указывает, что последовательно должны быть введены или выведены в соответствии с форматом *f* два действительных числа, являющиеся коэффициентами при действительной и мнимой частях комплексного числа. Например, после выполнения операторов:

опс (a,b) вещ комплексное;
a='2i'; b='-1+5i';
писать в_виде(a,b);

будет выведено

0.000000E+00+2.000000E+00I -1.000000E+00+5.000000E+00I

а после оператора

писать в_форме(a, b) (C(E (10, 3)), C(F(3)));

будет выведено 0.000E+00 2.000E+00 -1 5

а после выполнения операторов

get edit (a) (c(f(3))) (b) (c(e(6, 0)));
писать в_виде(a,b);

при наличии входного потока

«123␣ 45␣ ␣ ␣789-123 ␣ ␣ ␣ »

будет выведено 1.230000E+02+4.500000E+01I 7.890000E+02-1.230000E+02I

В рассматриваемой версии языка при вводе комплексных чисел оператором GET/ЧИТАТЬ по формату **C**, требуется, чтобы мнимая часть очередного числа во входном потоке обязательно была отделена от действительной части знаком «плюс» или «минус» независимо от числа пробелов между действительной и мнимой. Такое правило необходимо, если общая длина формата кратна 8. Например, при вводе по формату C(F(8)) во входном потоке недопустимо число « ␣1234567 ␣1234567», но допустимо число « ␣1234567 +1234567».

Элемент формата данных **P** будет рассмотрен в § 6.3.

Проверь себя

1. Что понимается под редактированием при вводе и выводе?
2. Каким образом устанавливается соответствие между элементами списка данных и элементами списка форматов?

3. Пусть *x*, *y* и *z* — элементы формата; запишите следующий список форматов без повторителей (x, 3 y, 2 (x, 2 (y, 2 z)))

4. Какая последовательность символов будет выведена после выполнения оператора писать в_форме('*') (A) (5, '101'Б) (A (6), B(2));

5. В каких случаях при выводе применяется элемент формата F?

6. Какой вид могут иметь данные во входном потоке при вводе с помощью элемента формата F?

7. В каких случаях при выводе применяется элемент формата E?

8. Какая последовательность символов будет выведена после выполнения оператора

писать в_форме (299, -299, 299) (e(11, 4), e(12, 4), e(11, 3));

9. Какой вид могут иметь данные во входном потоке при вводе с помощью элемента формата E?

10. В каких случаях применяется элемент формата C?

11. Включается ли в выходной поток буква I при выводе с помощью формата C (F (3))?

12. В каком виде представляются комплексные числа во входном потоке при вводе с помощью формата C?

13. Пусть переменные A и B описаны оператором

опс (A, B) вещ комплексное;

Каковы будут значения этих переменных после выполнения оператора

читать в_форме (a) (c(f(3)) (b) (c(e(5, 0)));

при условии, что входной поток имеет вид «-5␣ -10 ␣ 10e5 ␣ -12 ␣ ␣»

6.2. Управляющие элементы формата

Управляющие элементы формата предназначены для установки позиции, начиная с которой будет вводиться или выводиться очередной элемент данных. При вводе позиционирование может заключаться в пропуске определенного количества символов во входном потоке, в переходе к определенному символу относительно начала текущей или следующей строки, в пропуске определенного числа строк. При выводе позиционирование может заключаться в выводе определенного количества пробелов, в переходе к определенному символу относительно начала текущей или следующей строки, в выводе определенного числа строк, заполненных пробелами, в переходе к определенной строке относительна начала страницы и в переходе к новой странице. Позиционирование производится в тот момент, когда управляющий элемент формата встречается в процессе поиска в списке форматов очередного элемента формата данных. Например, при выполнении оператора

писать в_форме(A, B) (*a*, F(3), *b*) (C) (F(4));

где A, B и C — имена переменных; *a* и *b* — управляющие элементы формата, вначале будет выполнено позиционирование в соответствии с элементом *a*, затем выведено значение A, затем выполнено позиционирование в соответствии с элементами *b* и *a*, далее выведено значение B, а затем значение C. В промежутке между выводом значений B и C позиционирование в соответствии с форматом *b* не производится.

Имеется 6 управляющих элементов формата **COLUMN/СТОЛБЕЦ**, **LINE/ПЕРЕВОД_СТРОКИ**, **PAGE/ПЕРЕВОД_СТРАНИЦЫ**, **SKIP/С_НОВОЙ**, **TAB/ТАБУЛЯЦИЯ**, **X/П**.

Элемент формата **COLUMN/СТОЛБЕЦ(*n*)**, где *n* — целочисленная константа, задает переход к *n*-му символу вводимой строки или выводимой строки. Если при вводе *n*-й символ текущей строки уже введен, или при выводе *n*-й символ текущей строки уже выведен, то позиционирование производится к *n*-му символу следующей строки; иначе позиционирование производится к *n*-му символу текущей строки. Например, при исполнении оператора

```
get edit (a, b) (column (20), f(3));     читать в_форме (a, b) (столбец (20), f(3));
```

значение переменной A будет взято из 20 — 22 колонок очередной строки, а значение переменной B — из 20 — 22 колонок следующей за ней строки. При выполнении оператора

```
put edit (a,b) (column (10), a);     писать в_формет (a,b) (столбец (10), a);
```

значение переменной A будет напечатано с 10-й позиции очередной строки, а значение переменной B — с 10-й позиции следующей за ней строки. Для ключевого слова COLUMN допустимо сокращение COL. Например, последний оператор может быть записан в виде

```
put edit (a, в) (col (10), a);
```

Каждая строка, перед тем как в нее будут включаться выводимые символы, всегда заполняется пробелами. Поэтому части строк, оставшиеся незаполненными из-за применения элемента COLUMN, будут заполнены пробелами. Стандартная длина выводимой строки в рассматриваемых версиях языка равна 80 символам, но ее можно увеличить до 254 или сделать бесконечной, задав равной соответствующий параметр нулю. Средства языка, позволяющие изменять эту длину, рассмотрены в § 10.1.

Элемент формата **TAB/ТАБУЛЯЦИЯ** является разновидностью элемента COLUMN/СТОЛБЕЦ, но задает номер колонки кратным восьми, т.е. ТАБУЛЯЦИЯ(*n*), где *n =0, 1, 2, 3…,* задает позиции колонок соответственно 1, 8, 16, 24, …

Элемент формата **X/П(*n*)**, где *n* — целочисленная константа, задает пропуск я символов во входном или выходном потоке, рассматриваемом как непрерывная последовательность символов. Например, при выполнении оператора (если ранее ввода не было)

```
get edit (a, b) (f(3), x (3));          читать в_форме (a, b) (ч(3), п(3));
```

значение переменной A будет взято из 1 — 3-й колонок первой строки, а значение переменной B — из 7 — 9-й колонок этой же строки. Символы в 4 — 6-й колонках игнорируются. При выполнении оператора (если ранее вывода не было)

```
put edit (a, b) (f(5), x (2));          писать в_форме (a, b) (ч(5), п(2));
```

значение переменной A будет напечатано в 1 — 5-й позициях первой строки, а значение переменной B — в 8 — 12-й позициях этой же строки. Так как каждая строка перед началом ее вывода заполняется пробелами, то части строк, оставшиеся незаполненными из-за применения элемента формата X/П, будут заполнены пробелами.

Элемент формата **SKIP/С_НОВОЙ** или **SKIP/С_НОВОЙ(n)**, где *n* — целочисленная константа указывает, что при вводе или выводе надо перейти к началу *n*-й записи (строки) после текущей. Если выражение *n* не задано, то оно предполагается равным 1, т. е. в этом случае надо перейти к началу следующей записи (строки) после текущей. Например, если ранее ввода данных не было, то при выполнении оператора

```
get edit (a, b) (f(3), skip) (c, d) (skip (2),   Читать в_форме (a, b) (ч(3), с_новой)
f (3));                                          (c, d) (с_новой(2), ч (3));
```

значения переменных A, B, C и D будут взяты из 1 — 3-й колонок строк с номерами, соответственно, 1, 2, 4, 6. Следует учитывать, что если формат SKIP/С_НОВОЙ применить в самом начале ввода данных, то ввод начинается со 2-й строки. Символы входного потока, игнорируемые в соответствии с форматом SKIP/С_НОВОЙ, могут быть произвольными.

После выполнения оператора

```
put edit (a, b) (f (3), skip) (c, d) (skip   писать в_форме(a, b) (ч(3), с_новой)
(2), f (3));                                 (c, d) (с_новой(2), ч(3));
```

значения переменных A, B, C и D будут напечатаны в начале строк с номерами, соответственно, 1, 2, 4, 6 (при условии, что ранее вывода данных не было).

Обратите внимание, что если формат SKIP/С_НОВОЙ стоит последним в списке форматов, то он не сработает (т.е. не завершит ввод/вывод строки), поскольку к этому моменту список вводимых/выводимых элементов уже закончился, а, следовательно, все еще оставшиеся форматы, в том числе и

управляющие, будут проигнорированы. В таких случаях можно или задать следующий отдельный оператор завершения строки в вида

```
put skip;          или          писать с_новой;
```

или в рассматриваемой версии языка поместить формат SKIP/С_НОВОЙ прямо в список выводимых элементов, например

```
put edit(skip,a,skip,b,skip)(f(3));          писать в_форме (с_новой, a,
                                             с_новой, b, с_новой)(ч(3));
```

и тогда значения A и B будут выведены на отдельной строке каждая. Следствием такой возможности является запрет называть свою выводимую переменную именем SKIP или С_НОВОЙ.

Элемент формата **PAGE/ПЕРЕВОД_СТРАНИЦЫ** используется только при выводе данных, предназначенных для печати. Он указывает, что последующие данные надо выводить с начала очередной страницы. Обычно при этом выдается специальный символ, обозначающий конец текущей страницы. Во время выполнения программы всегда автоматически подсчитывается количество строк, выведенных на текущей странице, и когда это количество достигнет определенной величины, может быть вызвана некоторая программа, печатающая, например, заголовок страницы. Вопросы, связанные с использованием этой возможности языка, рассмотрены в гл. 10.

Элемент формата **LINE/ПЕРЕВОД_СТРОКИ(n)**, где n — выражение с целочисленным значением, также используется только при выводе данных, предназначенных для печати. Он указывает, что последующие данные надо выводить с начала n-й строки текущей страницы (действия, предпринимаемые в случае, если n-я строка текущей страницы уже хотя бы частично заполнена, рассмотрены в § 10.4).

В тех случаях, когда действия, задаваемые элементами формата SKIP/С_НОВОЙ(n), SKIP/С_НОВОЙ или PAGE/ПЕРЕВОД_СТРАНИЦЫ должны быть выполнены в самом начале работы оператора ввода или вывода, удобнее указывать эти элементы формата не в списке форматов, а в качестве самостоятельных конструкций оператора. Например, следующие два оператора функционально эквивалентны:

```
put edit (x) (skip, f (3));          писать в_форме (x) (с_новой, ч (3));
put skip edit (x) (f (3));           писать с_новой в_форме (x) (ч (3));
```

В одном операторе вывода может присутствовать только одна конструкция SKIP/С_НОВОЙ или SKIP/С_НОВОЙ(n) и одна конструкция или PAGE/ПЕРЕВОД_СТРАНИЦЫ. Конструкции SKIP/С_НОВОЙ(n), SKIP/С_НОВОЙ или PAGE/ПЕРЕВОД_СТРАНИЦЫ могут быть указаны не только

в операторах ввода и вывода с ключевым словом EDIT/В_ФОРМЕ, но и во всех других вариантах операторов GET/ЧИТАТЬ и PUT/ПИСАТЬ. Правила задания и использования этих конструкций во всех случаях идентичны рассмотренным выше. Например, после выполнения оператора

писать перевод_страницы в_виде (с_новой(2), 'ОЧЕРЕДНАЯ␣СТРАНИЦА');

фраза ОЧЕРЕДНАЯ СТРАНИЦА будет напечатана с начала 3-й строки очередной страницы.

В заключение параграфа рассмотрим несколько примеров. Пусть A — двумерный массив с диапазоном индекса по обоим измерениям от 1 до 6 и с числовыми значениями в диапазоне от -999 до 999. Требуется вывести значения элементов A «по строчкам» с двумя дробными цифрами. Для этого достаточно выполнить оператор

```
put edit (a) (skip, 6 f(8, 2));          писать в_форме (a) (с_новой, 6 ч(8, 2));
```

Так как изображение выводимого значения занимает в данном случае максимум 7 символов (5 цифр, точка и знак), а отведено под него 8 символов, то числа будут разделены, по крайней мере, одним пробелом.

Рассмотрим примеры печати таблиц. Пусть в процессе обработки массива X были установлены значения элементов массивов F, FD1 и FD2. Требуется вывести значения элементов всех массивов 4 столбца, над каждым из которых напечатано имя массива. Предположим, что массивы описаны оператором

```
dcl x(100) fixed, (f, fd1, fd2) (100)    опс x(100) точное, (f, fd1, fd2)(100)
float;                                   вещ;
```

Тогда искомый оператор вывода может, например, иметь вид

```
put skip edit('x', 'f', 'fd1', 'fd2')    писать с_новой в_форме('x', 'f', 'fd1', 'fd2')
(x(3), a, x(7), a, x(11), a, x(9), a)    (п(3), т, п(7), т, п(11), т, п(9), т)
((x(i), f(i), fd1(i), fd2(i)             ((x(i), f(i), fd1(i), fd2(i)
do i= 1 to n))                           цикл i= 1 to n))
(skip, f (6), 3(x(1), e(12, 5)));        (с_новой, ч (6), 3(п(1), e(12, 5)));
```

Пусть требуется напечатать таблицу значений функции «синус» с точностью 10^{-10} для углов от 0 до 90 градусов с шагом 1 градус. Таблица должна иметь вид

УГОЛ	СИНУС
nn	*n.nnnnnnnnn*

Таблица должна быть расположена, начиная с 50-й колонки строки. Данная задача может быть решена посредством следующей программы:

```
psin: proc main;                         psin: проц главная;
dcl i fixed(31);                         опс i точное(31);
```

```
put skip edit
('-'(20),'|угол|', ' синус','|', '-'(20) )
(2(col(50), a), 2 (x(4), a))
(('|', i, '|', sind( float(i,53)), '|', '-'(20)
do i = 0 to 90) )
(col (50), a, f(3), x(1), a,
f(13, 9), x(1), a, col (50), a);
end;
```

```
писать с_новой в_форме
('-'(20),'|угол|', ' синус','|', '-'(20) )
(2(столбец(50), т), 2 (п(4), т))
(('|', i, '|', sind( вещ(i,53)), '|', '-'(20)
цикл i = 0 до 90) )
(столбец(50), т, ч(3), п(1), т,
ч(13, 9), п(1), т, столбец(50), т);
конец;
```

П р о в е р ь с е б я

1. Для чего применяются управляющие элементы формата?
2. В какой момент времени выполняются действия, заданные управляющим элементом формата?
3. В каких случаях элемент формата COLUMN(*n*) задает переход к новой строке?
4. Предполагая, что ранее вывода не было, укажите, сколько пробелов будет размещено между значениями переменных A и B после выполнения оператора писать в_форме (A, B) (б(3), столбец(10));
5. Пусть значения элементов массива *A* представлены во входном потоке в виде целых десятичных констант, занимающих ровно 7 символов; все константы разделены одним символом, являющимся двоеточием. Составьте оператор ввода значений элементов массива *A*.
6. Какая последовательность символов будет выведена после выполнения оператора писать в_форме(2, 3) (ч(1),п(3)) (4) (ч(2));
7. Предполагая, что ранее ввода не было, укажите с какой по порядку строки, будет введено значение переменной D при выполнении оператора

   читать в_форме (A, B, C, D) (с_новой, ч(6), с_новой (2), ч(6));
8. Составьте оператор, печатающий значения элементов массивов *X* и *Y* таким образом, чтобы они были отделены друг от друга двумя пустыми строчками.
9. Какие действия определяют элементы формата PAGE/ПЕРЕВОД_СТРАНИЦЫ и LINE/ПЕРЕВОД_СТРОКИ(n)?
10. Какие элементы формата могут быть использованы в качестве самостоятельных конструкций операторов ввода и вывода?
11. Составьте оператор, который выводит «по строчкам» значения элементов двумерного массива H, описанного оператором опс H (8, 4) вещ (53);

12. Составьте программу, которая печатает таблицу значений функций $ln\ x$ и $log_2 x$ с точностью 10^{-8} для аргументов в диапазоне от 1 до 2 с шагом 0,01. Таблица, расположенная с 20-й колонки строки, должна иметь вид

X .	ДВ. Л О Г А Р И Ф М	НАТ. Л О Г А Р И Ф М
n.nn .	*n.nnnnnnnn*	*n.nnnnnnnn*

6.3. Элемент формата данных шаблон

Элемент формата **P** (русский эквивалент **Ш** - шаблон) представляет наиболее разнообразные возможности изображения выводимых числовых данных. В частности, он охватывает возможности, представляемые при выводе форматами F и E. В рассматриваемой версии языка этот элемент применяется только для вывода.

Элемент формата P в общем случае имеет вид

P'a' или *Ш'a'*

где *a* — шаблон, определяющий допустимое изображение данных. При вводе и выводе десятичных чисел в форме с фиксированной точкой шаблон обычно имеет вид $s_1 s_2 \ldots s_n \quad (n \geq 1)$ или

$$s_1 s_2 \ldots s_n \text{ V } s_{n+1} s_{n+2} \ldots s_{n+q} \quad (n \geq 0,\ q \geq 0,\ n + q \geq 1)$$

где s_i $(i = 1, 2, \ldots n + q)$ — символ шаблона, задающий допустимый вид i-го символа выводимой строки. Последовательность символов $s_1 s_2$ .. .s_n определяет изображение целой части числа, а последовательность символов $s_{n+1} s_{n+2} \ldots s_{n+q}$ — дробной части. Допускаются следующие символы шаблона чисел в форме с фиксированной точкой.

Символ шаблона **9** указывает, что соответствующий ему символ выводимой строки должен быть десятичной цифрой (пробел недопустим). Смотрите примеры 1 — 3 в табл. 6.1. Обратите внимание на то, что в примерах 2 и 3 десятичная точка не выводится и недопустима при вводе; для задания точки применяется специальный символ шаблона, рассмотренный ниже.

Символ шаблона **Z** указывает, что соответствующий ему символ выводимой строки должен быть либо пробелом, заменяющим нуль в начале числа, либо десятичной цифрой. Нули в дробной части числа заменяются пробелом только в том случае, когда изображение числа не содержит ненулевых цифр. Смотрите примеры 4 — 7 в табл. 6.1. Символ шаблона Z не может быть указан справа от символа 9; если в подполе дробной части шаблона указан хотя бы один символ Z, то символы 9 в шаблоне вообще

недопустимы. Таким образом, недопустимы шаблоны типа: 'Z9Z', '9VZZ', 'ZVZ9'.

Символ шаблона ***** аналогичен символу Z с той лишь разницей, что вместо пробела в изображении числа используется звездочка. Смотрите примеры 8 — 10 в табл. 6.1. Символы Z и * не могут быть заданы в одном шаблоне чисел в форме с фиксированной точкой. Недопустимы шаблоны типа: 'Z***', '**VZZ'.

Символы шаблона **S**, + и - применяются для указания позиции знака числа. Если в шаблоне чисел в форме с фиксированной точкой имеется ровно один символ знака, то соответствующий ему символ вводимой или выводимой строки должен быть: для неотрицательных чисел знаком «плюс» (символы шаблона S и +) или пробелом (символ шаблона -), для отрицательных чисел знаком «минус» (символы шаблона S и -) или пробелом (символ шаблона +). Смотрите примеры 11 — 17 в табл. 6.1. Одиночный символ знака может стоять либо слева от всех символов 9, Z или *, либо справа от них (пример 17); шаблоны типа '9S9' или 'Z-Z' недопустимы. Когда цифровые позиции в шаблоне заданы только символами Z либо только символами * и изображение числа не содержит цифр, то в позицию знака включается либо пробел, либо звездочка (см. примеры 18, 19).

Если в шаблоне числа в форме с фиксированной точкой имеется несколько однотипных символов знака, то вид соответствующих им символов вводимой или выводимой строки определяется по следующим правилам. Все эти символы, кроме самого левого, могут быть, по мере необходимости, значащими цифрами числа. Вид символа, стоящего в целой части числа слева от первой цифры, определяется по тем же правилам, что и для одиночного символа знака. Оставшиеся слева от него символы являются пробелами. Смотрите примеры 20 — 23 в табл. 6.1. Цифры дробной части числа заменяются на пробелы только в том случае, когда изображение числа не содержит ненулевых цифр (см. примеры 24, 25)

Если указано несколько символов знака, то все они должны быть однотипными, и при этом в шаблоне нельзя указывать символы Z и *. Все символы шаблона 9 должны быть в данном случае расположены справа от всех символов знака. Причем, если символы знака входят в подполе дробной части шаблона, то символы 9 вообще недопустимы. Таким образом, недопустимы шаблоны следующих типов: 'S-99', 'SSZZZ', ' V**', 'S9S','SVSS9'.

Хотя бы один из нескольких символов знака должен входить в подполе целой части шаблона; иначе знак не будет включаться в изображение числа (см. пример 26).

Для отделения дробной части числа от его целой части, для разбиения на группы цифр целой или дробной частей числа, для разделения чисел во

входном или выходном потоке применяются символы шаблона **B**, косая черта, запятая и точка.

Символ шаблона B задает позицию, в которой всегда размещается пробел; см. в табл. 6.1 примеры 27 — 29. Исключением является случай, когда все цифры числа заменены на звездочки; при этом и в позиции, соответствующей символу шаблона B, размещается звездочка (см. пример 28).

Символы шаблона косая черта, запятая и точка, называемые условными символами включения, задают позицию, в которой может размещаться, соответственно, косая черта, запятая и точка (см. примеры 30 — 33). Однако в ряде случаев в этой позиции размещается пробел, звездочка или знак числа.

Во-первых, если слева от условного символа включения, расположенного **в** подполе целой части шаблона, указан хотя бы один символ шаблона Z или * и во вводимой или выводимой строке слева от позиции, определенной условным символом включения, нет цифр, то в этой позиции размещается, соответственно, пробел или звездочка; см. примеры 34 — 36. Во-вторых, если в шаблоне имеется несколько символов знака и хотя бы один из них указан слева от условного символа включения, расположенного в подполе целой части шаблона, то при отсутствии цифр слева от позиции, заданной условным символом включения, в этой позиции размещается такой символ, как если ей соответствовал в шаблоне символ знака; см. примеры 37 — 41. В-третьих, если все цифровые позиции в шаблоне заданы только символами знака, либо символами Z, либо символами * и в изображении числа нет цифр, то в позиции, определенной любым условным символом включения, всегда размещается пробел или звездочка; см. примеры 42 — 44.

Символ шаблона $ задает позицию, в которой может размещаться денежный знак $; в ряде случаев в данной позиции размещается цифра, пробел или звездочка (см. примеры 45, 46). Правила применения символа шаблона $ идентичны правилам для символов знака, с той лишь разницей, что и для отрицательных, и для неотрицательных чисел в соответствующей позиции размещается символ $. В одном шаблоне допустимо и наличие символов знака, и наличие символов $; недопустимо лишь одновременное присутствие нескольких символов знака и нескольких символов $.

Пары символов шаблона **CR** и **DB** применяются при обработке финансовой информации для пометки отрицательных чисел (они обозначают, соответственно, «кредит» и «дебет»), В позициях, определенных этими символами шаблона размещаются для отрицательных чисел — пары символов DB или CR, а для неотрицательных чисел — пробелы (см. примеры 47 — 50). Если все цифры числа заменены на звездочки, то и в позициях, определенных символами DB или CR размещаются звездочки. Символы шаблона DB или CR

могут быть указаны только справа от всех других символов шаблона; при этом недопустимы символы шаблона S, +.

Если выводимое число содержит дробных цифр больше, чем допускается в изображении дробной части числа, то лишние цифры отбрасываются без округления (см. пример 51). Если выводимое число содержит больше значащих цифр целой части, нежели допускается в изображении числа, то происходит прерывание выполнения программы (прерывание может быть обработано при использовании средств, рассмотренных в гл. 8); см. примеры 52, 53.

ТАБЛИЦА 6.1

№ п/п	Шаблон	Значение числа	Изображение при выводе
1	999	25	025
2	9V9	0,2	02
3	V999	0,03	030
4	ZZZ	50	␣50
5	ZZ9	0	␣␣0
6	ZVZZZ	0,03	␣030
7	ZVZZZ	0	␣␣␣␣
8	**V**	3,5	*350
9	V**	0,05	05
10	V**	0	**
11	S99	0	+00
12	S99	-55	-55
13	-V99	0,2	␣20
14	-V99	-0,03	-03
15	+ZZ	5	+␣5
16	+ZZ	-16	␣16
17	9V9S	0,5	05+
18	SZZ	0	␣␣␣
19	-**	0	***
20	SSS9	28	␣+28
21	SSS9	-777	-777
22	---	-2	␣-2
23	+ + +	0	␣␣␣
24	SVSS	0,02	+02
25	SVSS	0	␣␣␣
26	VSSS	-0,2 -	200
27	9B999	1234	1␣234
28	*B*	0	***
29	*B*	1	**1
30	SZZ,VZZ	-5,8	␣5,80
31	SS.VSS	2,17	+2.17
32	/ZZ9/	53	/␣53/
33	Z.VZ	0,5	␣.5

№ п/п	Шаблон	Значение числа	Стандартное изображение числа при вводе и выводе
34	Z.V/Z	0,5	␣␣ /5
35	Z/ZZZ	238	␣␣ 238
36	*,V**S	—0,06	**06-
37	SS/SSS/	34	␣␣␣ +34/
38	SS/SSS.	234	␣ ␣+234
39	--,--	—4	␣␣␣ -4
40	--,--	—44	␣␣ -44
41	+ +.V+	0,5	␣␣ +5
42	ZV.Z	0	␣␣␣
43	/**/	0	****
44	,ZV, Z	0	␣␣␣␣
45	ZV.ZZ$	0,8	␣.08$
46	$$$9	24	␣ $24
47	99DB	28	28␣␣
48	99DB	—5	05DB
49	99CR	4	04 ␣␣
50	99CR	—37	37CR
51	9V.9	0,99	0.9
52	999	1234	234
53	SSS	234	+34

Проверь себя

1. Какую роль выполняет символ V в шаблоне чисел в форме с фиксированной точкой?
2. Какие символы в выводимом изображении числа могут соответствовать символу шаблона 9?
3. Пусть элемент формата имеет вид '9V9'; будет ли изображение числа, выводимого в соответствии с этим элементом формата, содержать:
   1) цифры целой части числа;
   2) цифры дробной части числа:
   3) точку?
4. Какая последовательность символов будет выведена после выполнения оператора: put edit (35, 2.8, 0.02) (p'999', p'99v99', p'v999');
5. В каких случаях пробел в выводимой строке будет соответствовать символу шаблона Z?
6. Какая последовательность символов будет выведена после выполнения оператора:

put edit (35, 2.8, 0.02, 0) (p'zzz', p'zzvzz', p'vzzz', p'zvz'); 5*

7. В каких случаях звездочка в выводимой строке будет соответствовать символу шаблона * ?

8. Какая последовательность символов будет выведена после выполнения оператора:

писать в_форме (35, 2.8, 0.02, 0) (p'***', p'**v**', p'v***', p'*v*');

9. С какой целью в шаблоне применяются символы S, + и -?

10. Какая последовательность символов будет выведена после выполнения оператора:

писать в_форме(99, 2.8, 0.02, -2, -0.5, -0,01, 0, 0, 0) (p'99s', p'zzvz + ', p'-v**');

11. В каких случаях пробелы в выводимой строке могут соответствовать нескольким символам знака, расположенным в подполе дробной части шаблона?

17- Какая последовательность символов будет выведена после выполнения оператора:

писать в_форме(23, -5.2, -0,02, 0) (p'sssvss');

18. С какой целью обычно применяются символы шаблона B / , .?

19. Какая последовательность символов будет выведена после выполнения оператора:

писать в_форме(128, 3, -3) (p'9в9в9', p'*b* *', p'-b-b-');

20. В каких случаях условному символу включения соответствует в выводимой строке: 1) звездочка; 2) знак «минус»; 3) пробел в дробной части числа?

21. Какая последовательность символов будет выведена после выполнения оператора:

писать в_форме(12345, 0.08, 0) (p'-/**, ***v.**');

22. Какая последовательность символов будет выведена после выполнения оператора:

писать _в_форме(2.8, 5, 6) (p'zzv.z$', p'$**', p'$$$$');

23. Какая последовательность символов будет выведена после выполнения оператора:

писать в_форме(28, 34, -82, -43) (p'$999db', p'$$$$cr');

24. Какие из следующих шаблонов недопустимы:

1) 9VZZ; 2) ZV99; 3) ***VZZ; 4) S**V99;
5) SSVS9; 6) SSS9V9;

6.4. Оператор удаленного формата

Имеется возможность задавать список форматов вне использующего его оператора ввода или вывода. Для этого применяется оператор вида

FORMAT *f* или **ВВЕСТИ_ФОРМАТ *f***

где *f* — произвольный заключенный в скобки список форматов. Перед данным оператором должна быть указана метка.

Для того чтобы в операторе ввода или вывода обратиться к списку форматов, заданному в каком-либо операторе FORMAT/ВВЕСТИ_ФОРМАТ, необходимо указать вместо списка форматов оператора ввода или вывода единственный элемент формата вида **R (*m*)**

где *m* — либо метка соответствующего оператора FORMAT/ ВВЕСТИ_ФОРМАТ, либо переменная со значением, равным метке данного оператора. Элемент формата R*(m)* может быть указан только один; перед ним не может быть задан повторитель. При этом просмотр списка форматов оператора ввода или вывода осуществляется таким образом, как если бы на месте этого элемента формата R*(m)* находился соответствующий ему список форматов оператора FORMAT/ВВЕСТИ_ФОРМАТ.

Оператор FORMAT/ВВЕСТИ_ФОРМАТ является невыполняемым; он может стоять как до, так и после использующих его операторов ввода или вывода. Однако и оператор FORMAT/ВВЕСТИ_ФОРМАТ, и все использующие его операторы ввода или вывода должны непосредственно входить в одну транслируемую единицу (блок или процедуру).

Рассмотрим пример. Имеются следующие сведения о деталях: наименование (16 символов), год начала выпуска (4 цифры), цена (до 999 рублей с копейками), масса в килограммах (до 99 килограмм с точностью до 100 грамм), ширина, длина и высота в метрах (до 5 метров с точностью до сантиметра); эти сведения надо ввести и распечатать в том же формате, что и при вводе, те сведения, которые относятся к деталям с годом начала выпуска не ранее 1967. Искомая программа может иметь вид

```
детали: проц  главная;
опс
имя          текст(16),
год          десятичное(4),
цена         десятичное(5, 2),
масса        десятичное(3, 1),
(l, s, h)    десятичное(3, 2);
когда конец_файла(стд_ввод) идти me;
```

```
m: читать          в_форме(имя, год, цена, масса, l, s, h) (r(f));
если год >=1967 тогда
писать с_новой в_форме(имя, год, цена, масса, l, s, h) (r(f))j
читать с_новой;
идти m;
f: ввести_формат(т(16), ч(4), п(l), ч(6,2),п(l),ч(4,1), 3 (п(1), ч(4, 2)));
me: конец детали;
```

Использование в формате R переменных со значением типа метка проиллюстрируем следующим примером. Пусть необходимо распечатать значения одномерного массива числовых переменных вещественного типа. При этом если выводимое число x удовлетворяет условию $0{,}1 \le |x| \le 99999{,}9$, то его следует печатать в соответствии с форматом Ч(8,1), иначе число следует печатать в соответствии с форматом Ч(8,3). Фрагмент программы, в котором осуществляется требуемая печать значений элементов массива A, может иметь вид

```
опс M метка;
M1:ввести_формат(п(1), ч(8,1));
M2:ввести_формат(п(1), ч(8,3));
цикл i=верх_граница(A) до ниж_граница(A);
        если abs(A(i)) > 0.1e0 тогда M=M1;
                                иначе M=M2;
        писать с_новой в_форме(A(i))(R(M));
…
```

П р о в е р ь с е б я

1. В чем состоит смысл использования формата R? Когда его целесообразно применять?
2. Измените рассмотренную в данном параграфе программу ДЕТАЛИ таким образом, чтобы год во входном потоке и при печати изображался бы двумя цифрами.
3. В каждой строке входного потока подготовлены сведения об абитуриентах: фамилия (20 символов), год рождения, оценки за 3 экзамена и средний балл аттестата (с точностью до сотых). Составить программу, которая вводит указанные сведения и печатает в том же формате, что и при вводе, те сведения, которые относятся к абитуриентам с суммой оценок за экзамены и среднего балла аттестата не менее 16 (использовать в программе оператор FORMAT/ВВЕСТИ_ФОРМАТ).

7. СТРУКТУРЫ

7.1. Понятие структуры, описание, обращение к элементам структур

Переменные произвольных, возможно различных типов, а также массивы переменных могут быть объединены в поименованные совокупности, называемые структурами. На практике в структуры объединяются такие переменные и массивы, которые связаны друг с другом в соответствии с логикой программы. Например, если несколько переменных или массивов одновременно участвуют во вводе или выводе, либо передаются вместе в качестве аргументов процедур, то может оказаться полезным объединить их в одну структуру. Оправданность использования структур обусловливается наличием в языке PL/1 средств, позволяющих манипулировать структурой, как единым объектом; эти средства будут рассмотрены в § 7.2.

Структуры могут выступать в роли элементов некоторой объемлющей структуры; подобные внутренние структуры принято называть подструктурами. Данная возможность применяется, когда, например, в одних действиях целиком участвует вся объемлющая структура, а в других — лишь некоторая ее подструктура. Подструктуры могут быть элементами не только самой внешней структуры, но и элементами промежуточных подструктур.

Таким образом, структура может представлять собой сложное многоуровневое иерархическое объединение переменных (например, допускается до 255 уровней иерархии в структуре). Структуры, подобно переменным, могут быть объединены в массивы произвольной размерности. Все структуры, входящие в один массив, должны иметь идентичную организацию. В частности, если у таких структур есть элементы-массивы, то соответствующие друг другу границы индексов элементов-массивов различных структур должны быть идентичными. Массивы структур (подструктур) могут быть элементами объемлющих структур; в частности, они могут быть элементами структур, которые в свою очередь также объединены в массив.

Все структуры и массивы структур описываются в операторах DECLARE/ОПИСАНИЕ. Их описание в развернутой форме имеет вид

$$l_1 n_1 a_{11} a_{12} \ldots a_{1k1},\ l_2 n_2 a_{21} a_{22} \ldots a_{2k2},\ \ldots,\ l_m n_m a_{m1} a_{m2} \ldots a_{mkm}$$

$$m \geq 2 \quad k_i\ (i=1, 2, \ldots, m) \geq 0$$

Здесь a_{ij} $(i=1, 2, \ldots, m;\ j=1, 2, \ldots, k_i)$ — описатель; n_i — идентификатор, являющийся именем структуры, массива структур или элемента структуры; l_i — положительное целое десятичное число без знака, называемое номером уровня структуры или элемента структуры.

С помощью номеров уровня задается внутренняя организация структуры. А именно, номер уровня структуры (подструктуры) или массива структур (подструктур) должен быть меньше номера уровня любого его (ее) элемента. Описания имен элементов структуры всегда должны следовать непосредственно за описанием имени структуры. Концом описания всех элементов некоторой структуры (подструктуры) является либо конец оператора, либо описание переменной или массива переменных, не входящих ни в какую структуру (т. е. описание без номера уровня), либо описание с номером уровня, не большим, чем у данной структуры (подструктуры). Номер уровня самой внешней структуры или массива структур всегда должен быть равен 1 (т. е. $l_1 = 1$).

Рассмотрим примеры. Пусть структура X состоит из трех элементов: переменных A и B со стандартными характеристиками и одномерного массива M с границами индекса от 1 до 10, состоящего из переменных с вещественными двоичными числовыми значениями с фиксированной точкой и точностью (31). Описание структуры X может иметь вид

```
1 X,
   2 A  вещ,
   2 B  вещ,
   2 M(10) точное(31);
```

Формально можно было бы использовать и другие номера уровней элементов структуры X, Например:

```
1 X,
   3 A  вещ,
   3 B  вещ,
   3 M(10) точное(31);
1 X,
   5 A  вещ,
   4 B  вещ,
   2 M(10) точное(31);
```

Однако чаще всего на практике номера уровней, указываемые при описании структуры, совпадают с фактическим номером уровня элементов структуры. Другой пример. Оператор описания

```
опс 1 H(5),
        2 X вещ,
        2 S,
          3 M точное,
          3 N(10) точное,
        2 Z(10) вещ;
```

определяет массив структур H; структуры, входящие в этот массив, состоят из трех элементов: переменной X, подструктуры S и массива переменных Z. Подструктура S состоит из двух элементов: переменной M и массива переменных N.

Описатели, применяемые при описании структур, массивов структур и их элементов, могут следовать в любом порядке, при условии соблюдения рассмотренных в предыдущих главах требований; в частности, описатель измерения всегда должен быть первым в списке описателей. Часть рассмотренных ранее описателей, а именно описатели EXTERNAL/ОБЩЕЕ, INTERNAL/МЕСТНОЕ и AUTOMATIC/ВРЕМЕННОЕ, недопустимы для элементов структур, они могут быть указаны лишь для имени самой внешней структуры или массива структур. С другой стороны, описатели типа значений недопустимы ни для структур (подструктур), ни для массивов структур (подструктур) даже в том случае, когда структура состоит только из переменных данного типа. При описании структуры, так же, как и при описании простых переменных и массивов переменных, можно выносить общие описатели за скобки. Например, оператор описания

опс 1 S(100), 2 A(100) вещ, 2 B(100) вещ; эквивалентен оператору

опс (1 S, 2 A вещ, 2 B вещ) (100);

Кроме описателей за скобки с левой стороны можно выносить и идентичные номера уровней элементов структур. Например, оператор описания

```
опс 1 S,
      2 A вещ,
      2 B текст(10),
      2 C точное,
```

эквивалентен оператору опс 1 S, (2(A вещ, B текст(10), C точное);

При описании нескольких структур с идентичной организацией удобно использовать описатель **LIKE/КАК** *a*, где *a* — неиндексированное имя структуры, подструктуры, массива структур или массива подструктуры (образование имен подструктур и массивов подструктур рассмотрено ниже).

Описатель LIKE/КАК *a*, будучи приписанным имени структуры A (подструктуры, массива структур или подструктур), указывает, что данная структура состоит из элементов с такими же именами и прочими характеристиками, что и структура *a* Например, оператор описания, определяющий 3 структуры — Al, A2 и A3

```
опс  1 A1, 2 (X, Y, Z(10)) точное,
       1 A2, 2 (A, B) вещ(53),
             2 SA2, 3(X, Y, Z(10)) точное,
```

```
        2 (M, N) вещ,
    1 A3,2(X, Y,Z(10)) точное;
```

Эквивалентен оператору

```
опс  1 A1, 2 (X, Y, Z(10)) точное,
     1 A2, 2 (A, B) вещ(53),
         2 SA2 как A1,
         2 (M, N) вещ,
     1 A3 как A1;
```

Номера уровней элементов структуры a, при перенесении их в описание структуры с описателем LIKE/КАК *a*, автоматически корректируются, что видно из приведенного выше примера описания структуры A2. Структура, определенная с помощью описателя LIKE/КАК *a* не может содержать элементы помимо тех, которые указаны в структуре *a*. На структуру, определенную с помощью описателя LIKE/КАК *a*, не переносятся описатели, приписанные имени структуры *a*. Например, оператор описания:

```
опс 1 sm(10),2(a, b, c(20)) вещ, 1 s как sm;
```

определяет массив структур SM и отдельную структуру S (уже не массив) с той же организацией, что и у элементов массива SM. Этот оператор эквивалентен следующему:

```
опс 1 sm(10), 2(a, b, c(20)) вещ, 1 s,2(a, b, c(20)) вещ;
```

Описатель LIKE/КАК допустим для структур a_2, указанных выше в каком-либо описателе LIKE/КАК a_2, но недопустим для подструктур, входящих в a_2, и для подструктур, непосредственно следующих за a_2, если a_2 — подструктура.

Идентификаторы, используемые в качестве имен структур, массивов структур и их элементов, могут быть, вообще говоря, произвольными. Однако должны соблюдаться следующие требования. Имя самой внешней структуры или массива структур не должно совпадать с описанными в этом же блоке или процедуре именами других внешних структур и массивов структур, именами простых переменных и массивов переменных, именами меток. Имя любого элемента структуры может совпадать с любым другим именем, включая имена, описанные в этом же блоке или процедуре; но у структуры не может быть два непосредственно входящих в нее элемента с одинаковым именем. Если идентификатор, являющийся именем элемента структуры, не используется для описания каких-либо других объектов в данном блоке или процедуре, то для обращения к элементу структуры можно использовать обычные имена, аналогичные именам простых переменных, массивов и элементов массивов. Например, если структура X описана оператором

```
опс 1 X, 2 A вещ, 2 B (10) вещ;
```

и идентификаторы A и B для других целей в данном блоке или процедуре не используются, то оператор писать в _виде (A, B (5));
задает вывод значений переменной A и 5-го элемента массива B, где A и B — элементы структуры X. Однако, если идентификатор, являющийся именем элемента структуры, используется в данном блоке или процедуре и для других целей, то для обращения к элементу структуры требуется составное имя вида

$$n_1p_1.n_2p_2. \ \ldots \ .n_kp_k \qquad (k \geq 2)$$

Здесь n_k — собственный идентификатор элемента структуры; n_i $(i = 1, 2, \ldots, k - 1)$ — идентификатор, являющийся именем структуры (подструктуры) содержащей элемент n_k; p_i $(i = 1, 2, .., k)$ — пусто или подсписок индексов вида

$$(p_{i1},p_{i2}, \ldots, \ p_{imi}.) \ (m \geq 1).$$

где p_{ij} $(j = 1, 2, \ldots, m_i)$ — индексное выражение. Идентификаторы n_1, n_2, ..., n_{k-1} должны следовать в порядке вложенности обозначаемых ими структур (массивов структур), т. е. структура с именем n_1 должна содержать структуру с именем n_2 и так далее. Если n_1 — имя самой внешней структуры, n_i $(i = 2, 3, .., k)$ — имя элемента, непосредственно входящего в структуру с именем n_{i-1}, то в этом случае составное имя называется *полным*. Использование полного составного имени для обращения к элементу структуры гарантирует правильность трактовки этого обращения при трансляции программы. Однако во многих случаях бывает достаточно использовать неполные составные имена, в которых имена части структур, содержащих данный элемент, опущены. При этом необходимо обеспечить лишь, чтобы данное неполное составное имя, независимо от списка индексов, было неприменимо к другому элементу этой или иной структуры, описанной внутри того же блока или процедуры, что и структура, к которой производится обращение. Например, рассмотрим следующий фрагмент программы (предполагается, что в ней нет других операторов описания):

```
a: проц главная;
опс 1 ss, 2 s, 3(x, y(10)) вещ, 2 ss2 точное;
…
b: блок;
опс 1 h, 2 a вещ, 2 z вещ, 2 s, 3(x, y, z) вещ, 2 hh, 3(x, r) вещ;
опс y(10, 10) вещ;
…
конец b; конец a;
```

Внутри процедуры B идентификатор Y используется и для обозначения элемента структуры H, и для обозначения отдельного двумерного массива. Поэтому для обращения к элементу Y структуры H необходимо использовать составное имя: H.S.Y (полное составное имя), или H.Y, или S.Y. Для обращения, например, к 5-му элементу массива Y, входящего в структуру SS, внутри процедуры A также необходимо использовать составное имя, например, SS.Y (5). Однако обращение S.Y(5) было бы внутри процедуры B не допустимо, несмотря на наличие списка индексов, так как это имя применимо и к элементу Y структуры H. Для обращения к элементу X подструктуры S структуры H следует использовать имя S.X или H.S.X. Имя H.X недопустимо ни для обращения к этому элементу структуры, ни для обращения к элементу X подструктуры HH, так как оно является неполным составным именем обоих этих элементов. Для обращения к элементу Z подструктуры S структуры H следует использовать имя H.Z.S или S.Z. Имя H.Z для этих целей не подходит, так как оно является полным составным именем элемента Z, непосредственно входящего в структуру H, и поэтому может быть использовано только для обращения к данному элементу.

Все индексные выражения, необходимые для обращения к некоторому элементу структуры, при использовании составного имени из k идентификаторов могут быть произвольным образом разбиты на m подсписков, где $1 \leq m \leq k$. При этом должен быть сохранен лишь порядок индексов. Например, если описан массив структур:

опс 1 M(10), 2 (A, B (100)) вещ;

то для обращения к 5 элементу массива B, входящего в 4 структуру массива M, кожет быть использовано любое из следующих составных имен:

M(4).B (5), M.B(4, 5) и M(4, 5).B.

Если идентификатор B в данном блоке или процедуре для других целей не используется, то для обращения к указанному выше элементу применимо и имя B(4, 5). Обратите внимание на то, что фактически имя B или M.B является именем двумерного массива переменных, аналогично имя A или M.A является именем одномерного массива переменных.

Входящие в структуру (или массив структур) переменные, массивы переменных и их элементы могут быть использованы практически во всех конструкциях языка PL/1, в которых допустимо использование отдельных переменных, отдельных массивов переменных или их элементов.

П р о в е р ь с е б я

1. Что называется структурой в языке PL/1? Подструктурой?
2. Что может являться элементом структуры?

3. Какие требования предъявляются к структурам, входящим в один массив структур?

4. Как изображается и интерпретируется номер уровня в описании структуры?

5. Укажите, из каких элементов состоит каждая из описанных ниже структур или подструктур

1) опс 1 S, 2 A вещ, 2 B вещ, 3 X вещ, 3 Yвещ, 3 Z вещ, 2 M точное, 2 N вещ;

2) опс 1 H(10), 2 HA вещ, 2 HS(50), 3 HX вещ, 3HY вещ, 2HB вещ;

3) опс 1 T(5), 2 TA вещ, 2 TS (10), 3 KS ,5 KA вещ, 4 KB вещ, 3 KX вещ, 2 TX вещ;

6. Составьте описание одномерного массива структур с границами индекса от 0 до 9; структуры, входящие в этот массив, состоят из 3 элементов: переменной A с вещественными двоичными значениями с фиксированной точкой и точностью (15), подструктуры B и одномерного массива C переменных с битово-строчными значениями длиной 8 битов (диапазон индекса массива C от -2 до 5); подструктура B состоит из двух переменных P и R с символьно-строчными значениями изменяющейся длины, их максимальная длина — 100 символов.

7. Допустимы ли описатели EXTERNAL/ОБЩЕЕ и INTERNAL/ МЕСТНОЕ при описании элементов структуры?

8. Пусть структура S состоит только из переменных с комплексными числовыми значениями; применим ли описатель COMPLEX при описании имени структуры S?

9. Запишите в наиболее краткой форме оператор описания

опс 1 STR, 2 X(10) точное, 2 V(10) точное, Z(10) точное;

10. Для чего применяется описатель LIKE/КАК?

11. Составьте оператор описания, эквивалентный приведенному ниже, но не содержащий описателей LIKE/КАК

```
опс 1 Sl (3), 2 ( X, Y (10), Z ) точное,
      1 S2,  2 (M, N) вещ, 2 SA2 как S1, 2 (P,R) точное,
      1 S3 как S1;
```

12. Может ли структура содержать элементы, собственные имена которых идентичны?

13. Что называется полным составным именем?

14. Пусть задан фрагмент программы

```
AA: проц главная;
опс 1 HX, 2 HY вещ,2 AR,3(A(10), B) вещ;
…
A: проц;'
опс 1 H, 2 HH, 3 (A, R) вещ, 2 C вещ, 2 AR, 3(A, B, C) вещ;
опс B вещ;
…
конец A; конец AA;
```

Предполагая, что в программе отсутствуют другие оператора описания и метки, перечислите все имена, допустимые внутри процедуры для обращения к следующим элементам структур:

1) элемент A подструктуры AR структуры H;
2) элемент B структуры H;
3) элемент C подструктуры AR структуры H;
4) элемент C, непосредственно входящий в структуру H;
5) элемент A подструктуры HH структуры H;
6) 5-й элемент массива A, входящего в структуру HX.

15. Пусть описан массив структур

```
опс 1 A (10), 2 E вещ, 2 C (10), 3 D вещ, 3 E (10) вещ;
```

Предполагая, что в программе отсутствуют описания других структур, перечислите все имена, допустимые для обращения к 6-му элементу массива E, входящего в 7-ю подструктуру массива C, который в свою очередь входит в 8-ю структуру массива A.

16. Пусть описан массив структур

```
опс 1 X (50), 2 (Y вещ, Z (20)), 3 (A, B (10)) вещ;
```

Укажите фактическую размерность массивов переменных с именами X.Y, X.Z.A, X.Z.B.

17. Составьте описание массива, содержащего сведения о наблюдениях за погодой в 1950 — 1981 гг.; для каждого года эти сведения включают общее количество осадков за год и максимальную, минимальную и среднемесячную температуру для каждого месяца года.

18. Составьте фрагмент программы, в котором на основе обработки массива, описанного в предыдущем упражнении, устанавливается и печатается среднегодовое количество осадков для тех лет, когда разница между максимальной и минимальной температурой каждого месяца не превышала 20 градусов.

7.2. Ввод и вывод структур и массивов структур

Имена структур (подструктур) и массивов структур (подструктур) могут быть указаны в списках данных операторов ввода (GET/ЧИТАТЬ) и операторов вывода (PUT/ПИСАТЬ).

Рассмотрим пример. Пусть имеется описание массива структур

```
опс 1 S(20, 40),
        2 A вещ,
        2 B,
          3(X, Y(10)) вещ,
        2 M (15) вещ;
```

Тогда оператор ввода `читать в_виде(S);` эквивалентен оператору вида

```
читать в_виде(((S(i1, i2) цикл i2 = 1 до 40) цикл i1 = 1 до 20));
```

Сравните порядок ввода или вывода структуры P и массива структур T, заданнных описанием

```
опс 1 P,
         2 (A, B) (100) вещ,
       1 T(100),
         2 (A, B) вещ;
```

Так при выполнении оператора `писать в_виде(P, T);` вначале будут выведены значения всех элементов массива P.A, затем всех элементов массива P.B, а затем пары значений элементов массивов T.A и T.B с одинаковыми индексами: T.A (1), T.B (1), T.A (2), T.B (2) и т.д.

Отметим, что при выводе значений элементов структур посредством оператора PUT/ПИСАТЬ с ключевым словом DATA/С_ИМЕНАМИ, всегда выводятся полные составные имена со списками индексов у «своего» имени.

Проверь себя

1. Можно ли в операторах ввода и вывода в списке данных указывать имена подструктур?
2. Пусть описан следующий массив структур:

```
опс 1 A (10, 20), 2(X, Y) вещ;
```

В каком порядке будут следовать в выходном потоке значения индексов элементов этого массива после выполнения оператора `писать с_именами(A);`?

3. Составьте программу, которая вводит массив сведений о деталях (шифр, цена, вес и год начала выпуска), упорядочивает его по возрастанию цены детали и печатает полученный массив.

7.3. Параметры-структуры

Процедуры могут иметь параметры, являющиеся структурами или массивами структур. Описываются такие параметры аналогично обычным структурам и массивам структур; допустимый вид описателей измерений и описателей длины в данном случае такой же, как и у параметров, являющихся переменными или массивами переменных.

Аргумент, сопоставляемый параметру-структуре, должен быть структурой. Элементы аргумента-структуры и параметра-структуры с одинаковым порядковым номером должны строго соответствовать друг другу так, как если бы они были самостоятельным аргументом и параметром.

Описание параметров, являющихся структурами или массивами структур, должно быть включено в описатель ENTRY/ДЛЯ_ВЫЗОВА в вызывающей процедуре.

Описание параметра-структуры в описателе ENTRY/ДЛЯ_ВЫЗОВА имеет вид

$$l_1a_{11}a_{12}\ldots a_{1n_1},l_2a_{21}a_{22}\ldots a_{2n_2},\ \ldots,\ l_ma_{m1}a_{m2}\ldots a_{mn_m}\ (m \geq 2\ \ n_i \geq 0)$$

где l_i $(i = 1, 2, \ldots, m)$ — номер уровня структуры или ее элемента ($l_1 \equiv 1$);
a_{ij} $(j = 1, 2, \ldots, n_i)$ — описатель элемента параметра-структуры. При этом должны быть описаны все элементы параметра-структуры. Значения номеров уровней элементов структуры, заданные в описателе ENTRY/ДЛЯ_ВЫЗОВА и в самой вызываемой процедуре, не обязаны совпадать; необходимо лишь, чтобы они одинаковым образом определяли организацию структуры. В описателе ENTRY/ДЛЯ_ВЫЗОВА нельзя выносить за скобки ни общие описатели, ни общие номера уровней.

Например, если внешняя процедура PS принимает в качестве параметра структуру вида

```
опс 1 A(100),
      2 FM текст(20),
      2 EXM(50),
        3 DT десятичное(6),
        3 OT точноет(31);
```

То в вызывающей процедуре эта внешняя процедура должна быть описана как

```
опс PS для_вызова (1 (100), 2 текст(20), 2(50), 3 десятичное(6), 3 точное(31);
```

Проверь себя

1. Какие требования предъявляются к аргументу, сопоставляемому параметру-структуре?
2. Какой вид имеет описание параметра-структуры внутри описателя ENTRY/ДЛЯ_ВЫЗОВА?
3. Могут ли номера уровней элементов аргумента-структуры не совпадать с номерами уровней соответствующих элементов параметра-структуры?
4. Составьте процедуру с одним параметром-структурой, содержащей сведения об одной детали (шифр, цена, год начала выпуска); процедура должна печатать сведения о детали в виде:

ДЕТАЛЬ: номер детали; ЦЕНА: цена детали

8. ПРЕРЫВАНИЯ ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА

8.1. Типы прерываний, условия возникновения прерываний

При выполнении программы могут возникать ситуации, требующие каких-либо особых действий, не предусмотренных в нормальной последовательности операторов, В основном такие ситуации возникают из-за ошибок, например, при делении на ноль, при преобразовании в число строки символов, имеющей недопустимый для данного преобразования вид, при выполнении арифметической операции, результат которой превышает допустимое значение. Возникновение подобных ситуаций приводит к прерыванию процесса выполнения программы, а затем выполняются действия, которые «обрабатывают» данное прерывание. Обработка прерывания может состоять либо из стандартных действий, предусмотренных языком PL/1, либо из действий, заданных программистом.

Рассмотрим некоторые из состояний, предусмотренных в языке PL/1.

При обработке чисел возможны следующие особые состояния вычислительного процесса.

Состояние **FIXEDOVERFLOW** или **ЧИСЛО_НЕ_ПРЕДСТАВИМО** возникает в тех случаях, когда результат арифметической операции или преобразования над числами в форме с фиксированной точкой при определенном масштабном множителе превышает допустимый максимум по количеству цифр — 15 цифр для десятичных чисел и 63 цифры для двоичных чисел. При этом фактический результат операции не определен. Для имени данного состояния допустимо сокращение **FOFL**.

Пусть, например, переменная A имеет вещественные десятичные значения с фиксированной точкой и точностью (15, 7) и пусть текущее значение переменной A равно 4. Тогда при выполнении оператора A=A*A; возникает состояние FIXEDOVERFLOW/ ЧИСЛО_НЕ_ПРЕДСТАВИМО, так как промежуточный результат операции умножения в соответствии с правилами языка PL/1 будет вычисляться с точностью, равной (15, 14), а значение этого результата равно 16.

Состояние **OVERFLOW/ЧИСЛО_ВЕЛИКО** возникает в тех случаях, когда порядок результата арифметической операции над числами в форме с плавающей точкой превышает допустимый максимум. При этом фактический результат операции не определен. Для имени данного состояния допустимо сокращение **OFL**. Пусть, например, переменная X имеет вещественные значения в форме с плавающей точкой и мантиссой 24 бита и пусть текущее значение этой переменной равно 10^{37}. Тогда при выполнении оператора

X=SQRT(X**2-4); возникает состояние OVERFLOW/ЧИСЛО_ВЕЛИКО, так как порядок промежуточного результата операции возведения в степень будет примерно равен 74, что больше допустимого максимума порядка.

Состояние **UNDERFLOW/ЧИСЛО_МАЛО** возникает в тех случаях, когда порядок ненулевого результата арифметической операции над числами в форме с плавающей точкой меньше допустимого минимума. При этом фактический результат операции устанавливается разным нулю. Для имени данного состояния допустимо сокращение **UFL**. Пусть, например, переменная X имеет вещественные значения в форме с плавающей точкой с мантиссой 53 разряда и пусть текущее значение этой переменной равно 10^{-160}. Тогда при выполнении оператора X=X**2; возникнет состояние UNDERFLOW/ЧИСЛО_МАЛО, так как порядок промежуточного результата операции деления будет равен -320, что меньше допустимого минимума. Отметим, что часто это состояние возникает при обращении к переменным со значениями в форме с плавающей точкой в тех случаях, когда эти значения ранее не были установлены.

Состояние **ZERODIVIDE/ДЕЛЕНИЕ_НА_0** возникает при делении на ноль для чисел любого типа (иногда оно возникает также вместо состояния FIXEDOVERFLOW/ЧИСЛО_ВЕЛИКО при преобразовании к двоичному основанию десятичных чисел в форме с фиксированной точкой). При этом результат операции деления не определен. Для имени данного состояния допустимо сокращение **ZDIV**. Пусть, например, текущее значение числовой переменной Z равно нулю. Тогда при выполнении оператора Z=Z*(1-2/Z); возникнет состояние ZERODIVIDE/ДЕЛЕНИЕ_НА_0 в момент деления 2 на Z. Отметим, что при обращении к числовым переменным с неустановленными ранее значениями (кроме переменных с десятичными значениями в форме с фиксированной точкой) их случайное значение часто оказывается равным нулю, поэтому использование такой переменной в качестве делителя может привести к возникновению этого состояния.

При работе с данными любого типа возможны следующие состояния общего характера.

Состояние **ERROR/ОШИБКА** возникает при ошибках в работе программы, для которых не предусмотрены специальные состояния, например, при недопустимом возведении в степень. Кроме того это состояние возникает в результате стандартной обработки прерываний для большинства других состояний (см. § 8.2).

Состояние **SUBSCRIPTRANGE/ВНЕ_ИНДЕКСА** возникает, когда при обращении к элементу массива значение индексного выражения выходит за

пределы допустимых границ диапазона индекса данного измерения. При этом результат обращения к элементу массива не определен. Для имени данного состояния допустимо сокращение **SUBRG**. Пусть, например, одномерный числовой массив A имеет диапазон индекса от 1 до 5. Тогда при выполнении оператора писать в_виде((A(I) цикл I=1 до 6)); возникнет состояние SUBSCRIPTRANGE/ВНЕ_ИНДЕКСА в момент обращения к элементу A(I) при I=6.

При работе со встроенными функциями SUBSTR/ПОДСТРОКА(*s, n, m*); может возникнуть состояние **STRINGRANGE/ВНЕ_ПОДСТРОКИ**, если выражения для позиции *n* или длина *m* выходят за пределы текущего значения строки.

Любое состояние вычислительного процесса может быть имитировано посредством оператора **SIGNAL/СИГНАЛ *a***, где *a* — имя состояния. После обработки прерывания, вызванного этим оператором, выполняется следующий за ним оператор программы, за исключением тех случаев, когда при обработке прерывания программистом явно задано иное продолжение выполнения программы. Управление не передается следующему оператору также при имитации состояния ERROR/ОШИБКА, т.е. при выполнении оператора SIGNAL ERROR; или СИГНАЛ ОШИБКА;

Проверь себя

1. Приведите примеры ситуаций, которые приводят к прерыванию вычислительного процесса.
2. В каких случаях возникают состояния:
   1) FIXEDOVERFLOW/ЧИСЛО_НЕ_ПРЕДСТАВИМО
   2) OVERFLOW/ЧИСЛО_ВЕЛИКО
   3) UNDERFLOW /ЧИСЛО_МАЛО
   4) ZERODIVIDE/ДЕЛЕНИЕ_НА_0
3. Какое из состояний вычислительного процесса возникнет при выполнении каждой из приведенных ниже последовательностей операторов?
   1) опс A dec(7, 7), B dec(3, 3); A =0.9; B = A* A* A;
   2) опс(X, Y) вещ; X, Y=1E-37; X = X*Y;
   3) опс(M, N) точное; M, N = 2; M = (M - 2)/(N - 2);
   4) опс(M, N) точное; N = 32000; M=2*N;
   5) опс(X, Y) вещ; X = 1E-37; Y = 1E37; X = Y/X;
   6) опс A dec(9, 0); A = 99999; A = A*A;
4. Приведите примеры ситуаций, которые приводят к возникновению состояния ERROR.

5. Укажите, при выполнении каких из приведенных ниже последовательностей операторов возникнет состояние SUBSCRIPTRANGE/ВНЕ_ИНДЕКСА.

опс A(5) вещ;	опс A(5) вещ;	опс A(5,4) вещ;
цикл i=1 до 6;	цикл i=1 до 6 с_шагом 2;	цикл i=1 до 5;
A(i)=i; конец;	A(i)=i; конец;	A(i, i)=i; конец;

6. Какой оператор позволяет имитировать возникновение любого состояния вычислительного процесса?

8.2. Обработка прерываний

Для каждого состояния в языке PL/1 предусмотрена определенная стандартная обработка. Для состояний, рассмотренных в предыдущем параграфе, эта обработка состоит в следующем.

Стандартная обработка прерывания для состояния ERROR/ОШИБКА заключается в печати диагностического сообщения и в переходе к завершению выполнения программы.

Стандартная обработка прерывания для состояний
SUBSCRIPTRANGE/ВНЕ_ИНДЕКСА,
STRINGRANGE/ВНЕ_ПОДСТРОКИ,
FIXEDOVERFLOW/ЧИСЛО_НЕ_ПРЕДСТАВИМО,
OVERFLOW/ЧИСЛО_ВЕЛИКО,
UNDERFLOW/ЧИСЛО_МАЛО,
ZERODIVIDE/ДЕЛЕНИЕ_НА_0
заключается в создании состояния ERROR/ОШИБКА с печатью соответствующего диагностического сообщения.

Программист задает собственную обработку прерываний посредством составного оператора вида **ON/КОГДА *a t***

где a — имя состояния; t — либо отдельный оператор (кроме PROCEDURE, BEGIN, DO, END, DECLARE, RETURN, FORMAT, ENTRY или ПРОЦЕДУРА, БЛОК, ЦИКЛ, КОНЕЦ, ОПИСАНИЕ, ВОЗВРАТ, ЗАДАТЬ_ФОРМАТ, ДЛЯ_ВЫЗОВА), либо блок; в обоих случаях в начале t не могут стоять метки. Именно оператор или блок t и будет выполнен при возникновении прерывания для состояния a.

Оператор ON/КОГДА является выполняемым оператором и должен быть выполнен до возникновения прерывания, обработка которого задается этим оператором. Непосредственно при выполнении оператора входящий в его состав оператор или блок t (см. выше) не выполняется.

В качестве примера рассмотрим программу, которая печатает значение максимального числа вида 2^m, где m — целое, представимого в памяти в форме с плавающей точкой. Искомая программа может иметь вид

```
F2M:проц главная;
опс X вещ(53);
когда число_велико идти OUT;
X=1e0;
M: X= X*2; идти M;
OUT: писать с_именами(X);
конец;
```

В данном примере оператор идти OUT; будет выполнен в тот момент, когда результат операции X*2 окажется непредставимым в форме с плавающей точкой.

Обработка прерывания, заданная оператором ON/КОГДА, будет выполняться только для прерываний, которые возникнут, начиная от момента выполнения данного оператора ON/КОГДА и кончая моментом выхода из блока или процедуры, в которой непосредственно расположен этот оператор. Однако если в указанный период времени для данного состояния будет выполнен другой оператор ON/КОГДА, то обработка последующих прерываний будет производиться в соответствии с этим последним оператором. При этом если второй оператор ON/КОГДА непосредственно расположен в том же блоке или процедуре, что и первый оператор, то второй оператор полностью аннулирует действие первого оператора. Иначе после выхода из блока или процедуры, в которой расположен второй оператор ON/КОГДА, обработка прерывания будет снова производиться в соответствии с первым оператором.

Для тех прерываний, на которые не распространяется действие ни одного оператора ON/КОГДА, всегда выполняется стандартная обработка. Изложенные правила проиллюстрируем следующим примером. В процессе выполнения программы

```
INTER: проц главная;
опс B (10) вещ;
опс (I, J, K, L, M) точное;
читать в_виде(I, J, K, L, M, B);
блок;
опс A(50) вещ;
A= 2;
Ml: A(I) = B(I);
M2: когда вне_индекса идти M4;
```

```
M3: A(J) = B(J);
M4: вызов P;
M5:A(K) = B(K);
M6: когда вне_индекса идти OUTj
M7: A(L) = B (L);
Писать в_виде(A);
конец;
M8:B(M) = 9;
OUT: писать в_виде (B);
P: проц;
M9:A(I + J) = B(I - J);
M10: когда вне_индекса идти M8;
M11: A (K+L) = B(K-L);
END;
END;
```

возможны прерывания при возникновении состояния SUBSCRIPTRANGE/ВНЕ_ИНДЕКСА. Если прерывание произойдет при выполнении оператора с меткой M1 или M8, то оно будет обработано стандартным способом, если прерывание произойдет при выполнении оператора с меткой M3, M9, или M5, то оно будет обработано в соответствии с оператором ON/КОГДА с меткой M2; если прерывание произойдет при выполнении оператора с меткой M11, то оно будет обработано в соответствии с оператором ON/КОГДА с меткой M10; наконец, если прерывание произойдет при выполнении оператора с меткой M7, то оно будет обработано в соответствии с оператором ON/КОГДА с меткой M6.

Имеется два вспомогательных оператора, которые могут находиться только внутри оператора ON/КОГДА. Первый из них **LEAVE ON** (или **ХВАТИТ КОГДА**) эквивалентен переходу оператором GOTO/ИДТИ на последний END/КОНЕЦ оператора обработки прерывания. Он применяется для выхода из оператора обработки без выполнения всех входящих в него операторов. Второй оператор — **RESIGNAL/РЕСИГНАЛ** также предназначен для немедленного выхода из оператора обработки прерывания, но в этом случае возобновляется поиск установленного ранее оператора ON/КОГДА с тем же именем прерывания. Если такого нет — возбуждается стандартная реакция системы на прерывание.

Дополнительным оператором, позволяющим управлять обработкой прерывания, является оператор вида **REVERT/ОТМЕНИТЬ *a***, где *a* — имя состояния. Этот оператор аннулирует действие последнего оператора ON/КОГДА, выполненного непосредственно в том же блоке или процедуре,

что и оператор REVERT/ОТМЕНИТЬ. После этого будет выполняться такая же обработка прерывания для состояния ON/КОГДА, как и при входе в блок или процедуру, непосредственно содержащую оператор REVERT/ОТМЕНИТЬ. В качестве примера рассмотрим следующую программу:

```
REV: проц главная;
опс (I, J, K, L) точное(31);
читать в_виде(I, J, K, L);
Ml: когда вне_индекса идти OUT;
блок;
опс A (100) вещ;
читать в_виде(A);
M2: A(J) =2;
M3: когда вне_индекса ресигнал;
M4: A(K) = 3;
Отменить вне_индекса;
M5: A(L) = 4;
L+=1;
конец;
OUT: писать в_виде(A);
конец;
```

Здесь при выполнении операторов с метками M2, M4, и M5 может возникнуть состояние ВНЕ_ИНДЕКСА. Если прерывание произойдет при выполнении оператора с меткой M2 или M5, то оно будет обработано в соответствии с оператором ON/КОГДА с меткой M1. Если же прерывание произойдет при выполнении оператора с меткой M4, то оно будет обработано в соответствии с оператором ON/КОГДА с меткой M3 (т. е. стандартным способом).

Если обработка прерывания, заданная программистом, заканчивается оператором перехода (GOTO/ИДТИ), то выполнение программы продолжается с точки, определенной этим оператором. Это правило распространяется и на обработку прерываний для состояния ERROR/ОШИБКА. При других завершениях обработки прерывания, в том числе и при завершении стандартной обработки, осуществляется так называемый нормальный возврат (если, конечно, не был выполнен оператор STOP/СТОП). Нормальный возврат после обработки прерывания для состояния ERROR/ОШИБКА состоит в переходе к завершению выполнения программы. При нормальном возврате после обработки прерываний для состояний

FIXEDOVERFLOW/ ЧИСЛО_НЕ_ПРЕДСТАВИМО,
OVERFLOW/ЧИСЛО_ВЕЛИКО,

UNDERFLOW/ЧИСЛО_МАЛО,
ZERODIVIDE/ДЕЛЕНИЕ_НА_0,
STRINGRANGE/ВНЕ_ПОДСТРОКИ
SUBSCRIPTRANGE/ВНЕ_ИНДЕКСА,

продолжаются прерванные вычисления непосредственно с той точки, где они были приостановлены.

При обработке любых прерываний могут оказаться полезными встроенные функции **ONCODE** и **ONLOC** (обе без параметров). Значение встроенной функции ONCODE является целым числом, идентифицирующим причину прерывания. Значение встроенной функции ONLOC равно строке символов, изображающей имя самой внутренней процедуры, при выполнении которой произошло прерывание

Проверь себя

1. Укажите, в чем заключается стандартная обработка прерывания для каждого из состояний, рассмотренных в § 8.1.
2. Посредством какого оператора программист задает собственную обработку прерываний (приведите примеры)?
3. Является ли оператор ON/КОГДА выполняемым оператором?
4. Составьте программу, которая печатает значение минимального числа вида 2^m, где m — целое, представимого в памяти в форме с плавающей точкой с точностью 53 разряда?
5. Пусть в программе имеется несколько операторов, задающих обработку прерывания для одного и того же состояния и пусть программа не содержит внутренних блоков или процедур; по какому правилу выбирается в данном случае обработка прерывания в момент его возникновения?
7. В каком случае обработка прерывания, заданная оператором ON/КОГДА в некотором блоке, будет использована для обработки прерывания в блоке, внешнем по отношению к данному?
8. В каких случаях применяются операторы вида

   ON *a* RESIGNAL; где *a* — имя состояния?

9. Каков эффект выполнения оператора REVERT/ОТМЕНИТЬ *a*, где *a* — имя состояния?
10. В каких случаях возврат после обработки прерывания считается нормальным?
11. Что происходит при нормальном возврате после обработки прерывания для состояния

1) число_мало; 2) ошибка; 3) вне_подстроки;

12. Каково значение встроенных функций ONCODE и ONLOC?

8.3. Прерывание при работе со строками

Состояние ERROR/ОШИБКА возникает при работе со строками символов при попытке преобразовать недопустимую строку символов в число или строку битов (при вычислении выражений, присваивании или вводе. Стандартная обработка прерывания для данного состояния заключается в создании состояния ERROR/ОШИБКА с печатью соответствующего диагностического сообщения.

При нестандартной обработке прерывания для таких случаев удобно использовать встроенную функцию **ONTERM**, которая возвращает номер последнего считанного элемента из списка элементов в операторах ввода GET LIST/ЧИТАТЬ В ВИДЕ или GET DATA/ЧИТАТЬ С_ИМЕНАМИ.

Состояние **STRINGRANGE/ВНЕ_ПОДСТРОКИ** (сокращенно — **STRG**) возникает при недопустимых значениях второго или третьего аргументов встроенной функции SUBSTR/ПОДСТРОКА или псевдопеременной SUBSTR/ПОДСТРОКА. Эти значения являются допустимыми в следующих случаях. Обозначим длину первого аргумента функции или псевдопеременной SUBSTR/ПОДСТРОКА через l, значение второго аргумента через k и значение третьего аргумента (если он задан) через n. Тогда должны быть выполнены условия

$$1 \le k \le \max(1,l\,) \text{ и } 0 \le n \le l\text{-}k + 1.$$

Стандартная обработка прерывания для состояния STRINGRANGE/ВНЕ_ПОДСТРОКИ заключается в создании состояния ERROR/ОШИБКА с печатью соответствующего диагностического сообщения.

При нормальном возврате после любой нестандартной обработки прерывания для данного состояния всегда вычисляется значение функции SUBSTR/ПОДСТРОКА или выполняется присвоение псевдопеременной SUBSTR/ПОДСТРОКА с автоматически скорректированными значениями второго или третьего аргументов. Эти скорректированные значения (обозначим их соответственно через k' и n') определяются по формулам

$$k' = \max(l'\,, \min(\mathrm{l},\, k)),$$

$$n' = \begin{cases} 0 \text{ при } k > l' \\ max\big(0, min(n\ +\ min(k, 1) -\ 1, l' - k + 1)\big) \text{ при } k\ \le l' \end{cases}$$

где k и n — исходные значения второго и третьего аргументов; l' — длина первого аргумента в момент окончания обработки прерывания (в процессе

обработки прерывания исходная длина первого аргумента может быть изменена).

Прерывания для состояния STRINGRANGE/ВНЕ_ПОДСТРОКИ по умолчанию запрещены. Для отмены этого запрета следует использовать специальный параметр трансляции (см. § 11.5).

Проверь себя

1. В каких случаях возникает состояние ОШИБКА при работе со строками?
2. В чем заключается стандартная обработка прерывания для таких случаев?
3. Что возвращает встроенная функция ONTERM?
4. В каких случаях возникает состояние STRINGRANGE/ВНЕ_ПОДСТРОКИ и в чем заключается стандартная обработка прерываний для данного состояния?

8.4. Состояния типа «контрольная точка»

При отладке программ часто бывает необходимо проследить за изменениями значений тех или иных переменных, проконтролировать прохождение определенных точек программы или вызов определенных процедур. Для этих целей удобно использовать обработку прерываний для состояний типа «контрольная точка». С точки зрения языка PL/1 это тоже состояние ERROR/ОШИБКА, однако оно возникает не вследствие каких-то реальных ошибок в программе, а в результате работы со специальной программой — интерактивным отладчиком. Стандартная обработка этого состояния в PL/1 — завершение программы и выдача диагностического сообщения. Однако при задействовании отладчика стандартная обработка состояния меняется на выход в специальный интерактивный режим отладки. Подробно это будет описано в главе 12.

Сейчас же отметим, что самому задавать в программе обработку этого состояния не требуется, но программист может подготовить текст программы к последующей отладке. Он может расставить дополнительные метки (на которые не будет перехода оператором GOTO/ИДТИ). В дальнейшем он может задавать состояние «контрольная точка», когда выполнение программы будет проходить через эти метки. Также с помощью встроенной функции UNSPEC/МАШ_КОД программист может сам расставить специальные коды в программе, при выполнении которых тоже возникнет состояние «контрольная точка».

Для создания этого состояния при вызове заданных процедур или при обращении к заданным переменным, причем отдельно при чтении, отдельно при записи, специальной подготовки программы не требуется, поскольку имена процедур и переменных известны и доступны программе-отладчику.

Отладка в интерактивном режиме гораздо продуктивнее подхода, когда в программе требуется заранее определять обработку тех или иных состояний, поскольку в этом случае можно задавать и отменять обработку по ходу выполнения программы и по результатам этого выполнения. При этом сам текст программы может оставаться неизменным.

Проверь себя

1. Для чего предназначена обработка состояния «контрольная точка»?
2. В чем заключается разница между стандартной обработкой состояния «контрольная точка» при включении и отключении программы-отладчика?
3. Какие дополнительные изменения может вносить программист в программу для облегчения отладки?

9. РАЗМЕЩЕНИЕ ДАННЫХ

9.1. Автоматическое, статическое и управляемое размещение данных

Выше было рассмотрено автоматическое размещение данных, при котором память под значения переменных отводится в момент входа в блок или процедуру, где непосредственно описаны эти переменные; освобождается отведенная таким образом память в момент выхода из указанного блока или процедуры. Следовательно, значения переменных с автоматическим размещением не могут быть использованы после выхода из блока или процедуры, в которой описаны эти переменные. Если переменные с автоматическим размещением описаны внутри рекурсивных процедур, то память под их значения будет отводиться при каждом очередном входе в процедуру, независимо от того, освобождена ли память, ранее отведенная под значения этих переменных. При этом непосредственно всегда доступны лишь значения переменных, расположенные в участке памяти, отведенном в последнюю очередь. При выходе из рекурсивной процедуры освобождается та память, которая была отведена под значения переменных с автоматическим размещением при входе, соответствующем данному выходу. Все сказанное выше относится и к переменным, описанным внутри блоков, расположенных в рекурсивных процедурах.

Память под значения переменных со статическим размещением отводится обычно в момент начала работы всей программы, независимо от расположения описания таких переменных; освобождается отведенная таким образом память в момент завершения выполнения всей программы. Переменным, массивам и структурам со статическим размещением приписывается описатель **STATIC/ПОСТОЯННОЕ**. Для структур этот описатель может быть приписан лишь их старшим именам. Описатель STATIC/ПОСТОЯННОЕ предполагается по умолчанию для переменных, массивов или структур с внешними именами (т. е. при наличии описателя EXTERNAL/ОБЩЕЕ); во всех остальных случаях описатель STATIC/ПОСТОЯННОЕ должен быть указан явно. Описатель STATIC/ПОСТОЯННОЕ, так же как и описатель **AUTOMATIC/ ВРЕМЕННОЕ**, недопустим для параметров. Существенным ограничением, снижающим возможности использования данных со статическим размещением, является то, что длины строк переменных со статическим размещением и выражения для границ диапазона индексов массивов со статическим размещением могут быть заданы только константами.

Описанные выше способы размещения данных обладают следующими недостатками: переменные (массивы, структуры) с автоматическим размещением не могут иметь внешних имен, а обязательное отведение (освобождение) памяти под их значения при каждом входе (выходе) в блок или процедуру может приводить к излишним затратам времени; допустимость только констант для границ массивов и длин строковых значений переменных со статическим размещением сужает область их применения, а невозможность освобождения памяти, занятой их значениями, до завершения выполнения программы может приводить к излишним затратам памяти.

Указанные недостатки отсутствуют при размещении, управляемом программистом. Память под значения переменных, массивов и структур с управляемым размещением отводится и освобождается только при выполнении специальных операторов языка PL/1. Таким переменным, массивам и структурам всегда должен быть явно приписан описатель **CONTROLLED/СТЕК** (допустимо сокращение **CTL**); причем для структур этот описатель может быть приписан только их старшим именам. Выражения для границ массивов и длин строковых значений переменных с управляемым размещением могут быть произвольными, их значения вычисляются непосредственно перед отведением памяти. Вместо выражений и в описателе измерений, и в описателе длины должна быть указана звездочка (определение границ или длин в данном случае рассмотрено в § 9.1 и § 13.2).

Оператор, применяемый для отведения памяти под значения переменных, массивов и структур с управляемым размещением, имеет в общем случае вид

ALLOCATE/ДАТЬ_ПАМЯТЬ s_1, s_2, ..., s_n $(n \geq 1)$,

где s_i $(i = 1, 2, ..., n)$ — либо спецификация одного управляемого размещения, либо спецификация одного базированного размещения (см. § 9.4). Спецификация управляемого размещения задает отведение памяти либо под одну переменную, либо под один массив, либо под одну структуру; при этом нельзя задать отведение памяти под часть массива или структуры. В простейшем случае такая спецификация состоит только из имени переменной, массива или структуры. Например, если имеется описание

```
опс (A вещ, B текст(100), Z(200) бит(8) CTL,
1 S CTL, 2(X(100) вещ, Y бит(8));
```

то для отведения памяти под указанные в нем переменные A и B, массив переменных Z и структуру S достаточно выполнить оператор:

```
дать_память A, B, Z, S;
```

При выполнении этого оператора будут вычислены объемы указанных в операторе ALLOCATE/ДАТЬ_ПАМЯТЬ переменных. Описатели измерений в

операторе ALLOCATE/ДАТЬ_ПАМЯТЬ берутся из соответствующего оператора DECLARE/ОПИСАНИЕ.

В описываемой версии языка описатели значений границ могут быть вычислены во время работы программы, но в этом случае в операторе DECLARE/ОПИСАНИЕ должны быть заданы звездочки и необходимы еще операторы специальной подготовки (см. § 13.2).

Освобождение памяти, отведенной под значения переменных, массивов или структур с управляемым размещением, производится посредством оператора, имеющего в общем случае вид

FREE/ВЕРНУТЬ_ПАМЯТЬ $s_1, s_2, \ldots, s_n;$ $(n \geq 1)$

где, s_i (i =1, 2, . . ., n) — либо имя переменной, массива или структуры с управляемым размещением, либо спецификация освобождения базированной памяти (см. § 9.4). Посредством оператора FREE/ВЕРНУТЬ_ПАМЯТЬ нельзя освободить память, занимаемую частью массива или структуры.

Если оператор ALLOCATE/ДАТЬ_ПАМЯТЬ применяется к переменной (массиву, структуре) с управляемым размещением, память под которую уже отведена, то под значения переменной отводится новый участок памяти; старый участок при этом не освобождается. Такая операция может быть повторена многократно и каждый раз после отведения памяти будет непосредственно доступен только новый участок памяти. Таким образом, одновременно могут существовать несколько поколений значений переменной, массива или структуры с управляемым размещением; при этом непосредственно доступно только последнее поколение значений (такая организация памяти называется *стеком).* В различных поколениях данных с управляемым размещением могут быть по-разному определены границы индексов массивов и длины строковых значений. Однако имеется возможность в последующем поколении определить точно такие же значения границ или длин, как и в предыдущем поколении.

Если оператор FREE/ВЕРНУТЬ_ПАМЯТЬ применяется к переменной (массиву, структуре), у которой размещено несколько поколений значений, то освобождается лишь память, отведенная под самое последнее поколение. Одновременно становится доступным предпоследнее поколение значений. Когда оператор FREE/ВЕРНУТЬ_ПАМЯТЬ применяется к переменной (массиву, структуре), под которую вообще не отведена память, то при этом возникает аварийное прерывание программы (создается состояние ERROR/ОШИБКА).

При работе с данными с управляемым размещением часто используется встроенная функция **ALLOCATION/ЕСТЬ_В_СТЕКЕ**. Обращение к этой функции имеет вид ALLOCATION/ЕСТЬ_В_СТЕКЕ(u), где u — имя

переменной, массива или структуры с управляемым размещением. Значение функции является строкой из одного бита, равной '1'Б, если отведена память хотя бы под одно поколение значений переменной (массива, структуры), и равной '0'Б в противном случае.

В качестве примера, иллюстрирующего работу с данными с управляемым размещением, рассмотрим следующую задачу. Требуется ввести заранее неизвестное количество строк символов и распечатать все эти строки, начиная с последней. Длина каждой строки может достигать 250 символов; при этом предполагается, что все строки вместе могут быть размещены в оперативной памяти. Искомая программа может иметь вид

```
ctl: проц главная;
опс s текст(*) рд стек;
когда конец_файла(стд_ввод) идти out;
m: дать_память s;
читать в_виде(s);
идти m;
out: цикл пока(есть_в_стеке(s));
писать с_новой в_виде(s);
вернуть_память s;
конец пока;
конец ctl;
```

Переменные, массивы и структуры с управляемым размещением, а также любые их части могут быть без каких-либо ограничений в качестве аргументов сопоставлены обычным параметрам. При этом посредством параметра будет доступно только самое последнее поколение значений аргумента.

Проверь себя

1. Перечислите различия между автоматическим и статическим размещением данных.
2. Какой описатель задает статическое размещение данных и в каких случаях этот описатель предполагается по умолчанию?
3. Укажите основные преимущества размещения данных, управляемого программистом, по сравнению с автоматическим и статическим размещением.
4. Какой описатель задает управляемое размещение данных; предполагается ли он когда-либо по умолчанию?
5. Каким образом освобождается память, отведенная под данные с управляемым размещением (приведите примеры)?

6. Что происходит при применении оператора ALLOCATE/ДАТЬ_ПАМЯТЬ к переменной, массиву или структуре с управляемым размещением, под значения которой память уже отведена?

7. Что происходит при применении оператора FREE/ ВЕРНУТЬ_ПАМЯТЬ к переменной, массиву или структуре с управляемым размещением, для которой память отведена под несколько поколений значений?

8. Что будет напечатано после выполнения следующего фрагмента программы, при условии, что перед его выполнением память под переменные A и B не отведена?

```
опс (a, b) точное стек;
дать_память a;
a = 5;
дать_память a;
a = 6:
писать в_виде(a, есть_в_стеке(a), есть_в_стеке(b));
```

9. Пусть во входном потоке представлено несколько групп чисел, причем перед каждой группой указано количество входящих в нее чисел. Составьте программу, которая вводит все указанные числа и печатает их по группам, начиная с последней; числа, входящие в одну группу, должны быть напечатаны в том же порядке, что и при вводе (указание: использовать возможность создания нескольких поколений значений числового массива с управляемым размещением).

10. Каким образом устанавливается связь между обычным параметром и сопоставленным ему аргументом, если аргумент является переменной, массивом или структурой с управляемым размещением и память отведена под несколько поколении значений аргумента?

9.2. Установка начальных значений переменных

В тот момент, когда отводится память под значения переменной, массива или структуры с любым типом размещения, сами значения обычно оказываются неопределенными (если не использованы средства, рассмотренные в данном параграфе) и обращение к этим значениям может привести к непредсказуемым последствиям. Однако в языке PL/1 предусмотрена возможность явно указать начальные значения переменных, которые будут присвоены им одновременно с отведением памяти. В частности, переменной может быть приписан описатель

INITIAL/ЗАДАТЬ (*e*)

где e — выражение, значение которого и используется в качестве начального значения переменной (для ключевого слова INITIAL допустимо сокращение INIT). Выражение e может иметь одну из следующих форм:

а) любая константа (в т. ч. имя метки или точки входа);

б) указатель функции без параметров, возвращающей значение типа «указатель».

В рассматриваемой версии языка присваивать начальные значения можно только переменным со статическим размещением в памяти. Заметим также, что задавать начальные значения переменным с автоматическим размещением в памяти при входе в блок или процедуру проще и нагляднее явным оператором присваивания.

В качестве простейшего примера рассмотрим программу, которая вводит ряд целых чисел и печатает их сумму; при этом для переменной, с помощью которой подсчитывается сумма, зададим нулевое начальное значение. Искомая программа может иметь вид

```
sum: проц главная;
опс(s постоянное задать(0), a) точное(15);
когда конец_файла(стд_ввод) идти out;
m: читать в_виде(a);
s+ = a;
идти m;
out: писать в_виде(s);
конец;
```

Тип значения выражения, задающего начальное значение, не обязан совпадать с типом переменной, значение которой устанавливается. Во всех случаях будут выполнены необходимые преобразования.

При задании начальных значений массивов переменных, а также переменных, входящих в массив структур, применяется описатель INITIAL/ЗАДАТЬ, имеющий вид

INITIAL/ЗАДАТЬ (s_1, s_2, …, s_m) ($m \geq 1$)

где s_i (i = 1, 2, ..., m) — либо спецификация одного начального значения, либо спецификация нескольких одинаковых начальных значений, либо повторяемый подсписок начальных значений. Спецификация одного начального значения может быть либо выражением в любой из форм, перечисленных выше для случая одной переменной, либо звездочкой, указывающей, что для соответствующего элемента массива установка начального значения не требуется.

Спецификация нескольких одинаковых начальных значений имеет вид *(k)s*, где *k* константа с целочисленным значением, называемое повторителем; *s* — любая спецификация одного начального значения (звездочка также допустима).

Спецификация нескольких значений эквивалентна последовательности из *k* одиночных спецификаций *s*.

Например, описатель задать(3, (3) 2, (2) *, (2) 5, 1) эквивалентен описателю задать (3, 2, 2, 2, *, *, 5, 5, 1).

При установлении соответствия между элементами списка начальных значений и элементами массива список просматривается слева направо, а элементы массива перебираются в том же порядке, как и при вводе или групповом присвоении; то есть вначале берется элемент массива с минимальными значениями всех индексов, затем последовательно увеличивается крайний справа индекс и так далее, до элемента с максимальными значениями всех индексов. Например, описание

опс A (3,3) вещ постоянное задать (1,2,3,4,5,6,7,8,9);

задает следующие начальные значения элементов массива: A(1,1)=1; A(1,2)=2; A(1, 3)=3; A(2, 1)=4; A(2, 2)=5; A(2, 3)=6; A(3, 1)=7; A(3, 2)=8; A(3, 3)=9. Если при установлении соответствия между элементами списка начальных значений и элементами массива список будет исчерпан раньше конца массива, то для оставшихся элементов массива начальные значения не устанавливаются. Если же конец массива будет достигнут раньше конца списка, то выдается сообщение об ошибке.

Описатель INITIAL/ЗАДАТЬ не может быть приписан именам структур, подструктур и именам массивов структур или подструктур. Однако он может быть приписан любым младшим элементам структур. Например, описание

```
опс
1 S постоянное,
  2 A вещ комплексное задать('5+2i'),
  2 M (10) точное задать ((10) 0),
  2 T(3),
    3 C текст(3) задать('X'(3), (2)'X'),
    3 H точное задать(*, 2, 2),
  2 N (10) точное задать(0);
```

задает следующие начальные значения элементов структуры S: S.A=5+2I; значения всех элементов массива S.M равны нулю; S.T.C (1)='XXX'; S.T.C (2)='X'; S.T.C(3)='X'; S.T.H(l) - значение не определено; S.T.H (2)=2; S.T.H (3)=2; S.N (1)=0; остальные элементы массива S.N не определены.

Описатель INITIAL/ЗАДАТЬ не допустим для обычных параметров.

Для переменных типа «указатель» (описание ниже) предусмотрена еще одна форма описателя

INITIAL/ЗАДАТЬ(*p*());

где *p* — имя входа в процедуру-функцию без параметров, возвращающую значение также типа указатель Подобный описатель указывает, что при любом обращении к такой переменной, происходит вызов функции *p* и ее значение становится текущим значением этой переменной.

Такой прием позволяет, например, обращаться к массиву, расположенному в несмежных участках памяти, поскольку при обращение к элементу массива по индексу (подразумевается, что при обращении используется и указатель), каждый раз вызывается функция *p*, которая по индексу вычисляет в какой участок памяти входит данный элемент массива.

Напоминаем, что цифра **1**, поставленная перед оператором группы, превращает эту группу операторов в еще один способ начальной инициализации, поскольку больше эта группа выполняться не будет, даже при многократном прохождении управления в программе через это место. Таким образом, можно создать любую, сколь угодно сложную инициализацию переменных, так как никаких ограничений на операторы такой группы нет.

Например, 1 {; x=f(a,b,c+10); }; переменная *x* при выполнении программы получит один раз значение функции *f*, вызванной с указанными параметрами.

Также в рассматриваемой версии языка имеется описатель **VALUE/ЗНАЧЕНИЕ**, который подобен описателям STATIC INIT или ПОСТОЯННОЕ ЗАДАТЬ, однако затем менять в программе эти переменные невозможно

Проверь себя

1. Каковы значения переменных при условии, что описатель INITIAL/ЗАДАТЬ не использовался?
2. Какими двумя формами может быть задано начальное значение в описателе INITIAL/ЗАДАТЬ?
3. Составьте программу, которая вводит ряд чисел и печатает их произведение; при этом для переменной, посредством которой подсчитывается произведение, задайте начальное значение с помощью описателя INITIAL/ЗАДАТЬ.
4. Обязан ли тип значения переменной совпадать с типом константы, задающего ее начальное значение?

5. Перечислите допустимые формы элементов списка начальных значений массива переменных.

6. Какие начальные значения примут элементы каждого из описанных ниже массивов:

1) опс A(4) текст(3) задать('H'((3), (3) 'HH');
2) опс B(3,3)вещ задать(9,8,7,6,5,4,3,2,1);
3) опс D(10) вещ задать(0);
4) опс C(50) вещ задать((50) 0);
5) опс E(12) вещ((2)*, (2) 3, (3) 4, *, 1).

7. Может ли быть описатель INITIAL/ЗАДАТЬ приписан имени структуры (подструктуры)?

8. Пусть имеется структура S, состоящая из трех элементов — числовых переменных A, B и C; составьте оператор описания, в котором для всех элементов структуры S задавались бы нулевые начальные значения.

9. Что задает описатель INITIAL/ЗАДАТЬ в операторе

опс X указатель постоянное задать(Y());

9.3. Определяемые переменные

В языке PL/1 имеется возможность описать различные переменные, массивы или структуры таким образом, чтобы их значения занимали один и тот же участок памяти. Тем самым изменение значения одной такой переменной будет автоматически приводить к изменению значений других переменных. В группе переменных (массивов, структур), размещаемых на одном участке памяти, имеется ровно одна переменная (массив, структура), называемая базовой. Базовая переменная описывается обычным образом, независимо от остальных переменных, связанных с отводимым ей участком памяти и называемых определяемыми. Определяемые переменные, массивы и структуры описываются посредством описателя **DEFINED/НА_МЕСТЕ(*u*)**, где *u* — имя базовой переменной, массива или структуры. Базовая переменная, массив или структура могут иметь автоматическое или статическое размещение, они могут быть параметрами независимо от типа размещения сопоставляемых аргументов. Описатель DEFINED/НА_МЕСТЕ не совместим с описателями типа размещения (AUTOMATIC/ВРЕМЕННОЕ, STATIC/ПОСТОЯННОЕ, CONTROLLED/СТЕК, BASED/ОСНОВА — см. § 9.4), он не может быть приписан параметрам. При описании определяемых структур описатель DEFINED/НА_МЕСТЕ должен и может быть приписан только старшему имени структуры. Для ключевого слова DEFINED допустимо сокращение **DEF**.

Следует отметить, что в тексте программы описания базовых переменных должны идти раньше описания соответствующих определяемых переменных.

Имеется разновидность описателя DEFINED/НА_МЕСТЕ(*и*), где в *и* входит не только имя базовой переменной, но и целочисленная константа со знаком плюс, которая задает *смещение* в байтах определяемой переменной относительно базовой. Например, в операторах описания

```
опс 1 S,
        2 A точное (15),
        2 B точное (15),
    X точное (15) на_месте(S+2);
```

Переменная X будет размещена в памяти там же, где и переменная S.B, поскольку переменная S.A в данном случае занимает в памяти два байта.

Обратите внимание, что никаких проверок соответствия типов или размеров базовых и определяемых переменных (массивов, структур) не производится. Используя этот описатель можно обращаться к произвольному участку памяти, обходя все имеющиеся проверки. Поэтому использовать определяемые переменные, следует с осторожностью, понимая размер каждого элемента. В общем случае размер базовой переменной не должен быть меньше соответствующих определяемых переменных.

Обычно описатель DEFINED/НА_МЕСТЕ используют, когда алгоритм требует обращения к одному и тому же объекту как к переменным разных типов без необходимости преобразований, например, при вычислении *циклически избыточного кода*, когда требуется переменную представлять то как строку бит, то как целое число, но преобразование не нужно, поскольку использование как строки бит, так и как числа производится в разных операторах алгоритма.

Также описатель DEFINED/НА_МЕСТЕ может использоваться для реализации в программе *адресной арифметики*, когда базовые переменные имеют тип указатель, а соответствующие определяемые переменные имеют тип двоичных целых чисел. В этом случае к указателям арифметические операции неприменимы, однако к соответствующим определяемым переменным — применимы.

Проверь себя

1. Какие переменные, массивы и структуры называются определяемыми?
2. Перечислите требования, предъявляемые к базовой и определяемой переменной.

3. Допустим ли следующий оператор описания и если нет, то укажите, почему

опс 1 S, 2 (A, B) вещ, 1 SD, 2 X вещ на_месте(S.B), 2 Y вещ на_месте(S.A);

4. Каковы будут значения переменной X после выполнения фрагмента программы

```
опс (X, Y на_месте(X)) вещ (24);
X=1;
Y=0;
```

5. Каковы будут значения элементов массива A после выполнения фрагмента программы

```
опс A (4) вещ, B (2:3) вещ на_месте(A);
A = 0;
B(3) = 2;
```

6. Как задается смещение в байтах для определяемой переменной относительно начала базовой переменной?

9.4. Базированные переменные

С каждым байтом оперативной памяти взаимно однозначно связана характеристика, называемая адресом данного байта. В языке PL/1 предусмотрены переменные, значениями которых могут быть адреса любых байтов оперативной памяти, такие переменные называются переменными типа *указатель.*

Описателем данных типа указатель является описатель **POINTER/УКАЗАТЕЛЬ**. Для данного ключевого слова допустимо сокращение **PTR** или **УКАЗ**. В языке PL/1 допустимы и процедуры-функции со значениями типа указатель; при их описании также применяется описатель POINTER/УКАЗАТЕЛЬ. К данным типа указатель не применимы никакие операции, кроме двух операций сравнения: «равно» (=) и «не равно (^=). Данные типа указатель не могут быть преобразованы в данные других типов.

Основное применение данных типа указатель состоит в том, чтобы определять (указывать) начало участка памяти, который предполагается отведенным в данный момент под переменные, массивы или структуры, называемые *базированными.* Такие переменные, массивы и структуры описываются посредством описателя имеющего обычно вид **BASED/ОСНОВА(*u*)**, где *u* — имя переменной типа указатель. Допустим указатель, состоящий только из ключевого слова BASED/ОСНОВА. Описатель BASED/ОСНОВА, являясь описателем типа размещения, несовместим с описателями AUTOMATIC/ВРЕМЕННОЕ, STATIC/

ПОСТОЯННОЕ, CONTROLLED/СТЕК, DEFINED/НА_МЕСТЕ. При описании базированных структур и массивов структур описатель BASED/ОСНОВА указывается только для их старшего имени. Описатель BASED/ОСНОВА не допустим для параметров. Имена любых базированных переменных, массивов и структур всегда являются внутренними и поэтому описатель BASED/ОСНОВА не совместим с описателем EXTERNAL/ОБЩЕЕ. В описателях измерений базированных массивов допустимы только целые десятичные константы.

За базированными переменными, массивами и структурами никогда не закрепляется никакой определенный участок памяти. При каждом конкретном обращении к базированной переменной (массиву, структуре) адресом начала отведенного под нее участка памяти считается текущее значение базового указателя данной переменной (массива, структуры). По умолчанию таким базовым указателем считается переменная, заданная в описателе BASED/ОСНОВА. Однако имеется возможность при любом конкретном обращении к базированной переменной, (массиву, структуре) явно задавать текущий базовый указатель. Для этого применяются имена вида

a -> *b*

где *a* — выражение, задающее значение текущего базового указателя; *b* — имя базированной переменной, массива или структуры, либо имя элемента базированного массива или структуры. Имена вида *a* -> *b* называются *именами с явно заданным указателем.* Если имя с явно заданным указателем применяется для обращения к элементу базированного массива или структуры, то значение указателя определяет начало участка памяти, условно отведенного под весь базированный массив или структуру, а не под элемент, к которому производится обращение.

При работе с указателями часто применяется встроенная функция **ADDR/АДРЕС(*u*)**, где *u* — имя переменной, массива или структуры. Значением функции ADDR/АДРЕС(*u*) является адрес начала участка памяти, отведенного в данный момент под переменную (массив, структуру) и; таким образом, встроенная функция ADDR/АДРЕС имеет значение типа указатель.

Рассмотрим пример. Пусть описана структура

```
опс 1 S,
      2(A, B, C) вещ,
      2 D текст (4),
      2 X(5) вещ;
```

и пусть описаны следующие базированные переменные, массивы и структуры:

```
DCL 1 SB  основа(PA),
```

```
      2(W, V) вещ,
    XB(3) основа(PB) вещ,
    Z     основа(PC) вещ,
    CB    основа(PC) текст(3),
(PA, PB, PC, PX) указ;
```

После выполнения последовательности операторов

```
PA = адрес(S);
PB = адрес(S.X(3));
PC = адрес(D);
PX= адрес(S.C);
SB.V,SB.W = 2;
XB(1),XB(2),XB(3) =3;
CB ='ABC';
PX->Z = 4;
```

элементы структуры S будут иметь следующие значения: A=2; B=2; C=4; первые три символа значения элемента D будут равны 'ABC', а последний символ значения D останется не определенным; значения последних трех элементов массива X будут равны 3, а значения первых двух элементов этого массива останутся не определенными.

Подчеркнем, что при присвоении значения базированной переменной никоим образом не учитывается, за какой переменной закреплен используемый при этом участок памяти. Поэтому допустимы, например, следующие фрагменты программы:

```
опс C текст(100),
B(10) основа(P) вещ;
P = адрес(C);
B = 0;
```

Здесь часть участка памяти, отведенная под значение переменной C, будет заполнена десятью нулевыми значениями в форме с плавающей точкой. Подобные возможности позволяют программисту весьма гибко использовать оперативную память. Однако использовать эти возможности языка PL/1 допустимо лишь после тщательного ознакомления с деталями отведения памяти под переменные, массивы и структуры с различными типами значений (см. § 13.5). На данном же этапе изучения языка рекомендуем составлять лишь такие программы, в которых базированные переменные, массивы или структуры имеют такую же организацию, как и переменные, массивы (или части массивов), структуры (подструктуры), за которыми закреплен участок памяти, адресуемый базовым указателем. Например, пусть описана структура S и массив структур M

```
опс 1 S основа(P),
        2 (N точное,
          X вещ),
      1 M(100) STATIC,
        2 (K точное,
          Z вещ),
      P указ;
```

Тогда после выполнения операторов

```
P= адрес (M(40)); S = 0;
```

элементам структуры M (40) будут корректно присвоены нулевые значения. Однако если бы в указанном выше операторе описания был описатель точное(31), такое присвоение уже не имело бы места.

При работе с указателями помимо функции ADDR/АДРЕС часто используется встроенная функция без параметров с именем **NULL/ПУСТО**. Значением этой функции является значение типа указатель, которое заведомо не равно никакому адресу в памяти. Встроенная функция NULL/ПУСТО может быть указана в описателе INITIAL/ЗАДАТЬ статических переменных типа указатель. Примеры использования встроенной функции NULL/ПУСТО будут даны ниже. Имеется также встроенная функция NULL_PTR/ПУСТОЙ_УКАЗ, которая всегда возвращает внутренний указатель со значением NULL/ПУСТО. Эта функция нужна в случаях вызова процедур с параметрами-указателями, когда такой параметр не требует аргумента. В этом случае поставить вместо аргумента NULL/ПУСТО нельзя, поскольку он имеет ни к чему не преобразуемый тип «адрес», а не указатель.

Под любую базированную переменную, массив или структуру можно отвести ранее свободный участок памяти соответствующего размера. Для этого используется рассмотренный в § 9.1 оператор ALLOCATE/ДАТЬ_ПАМЯТЬ, в котором наряду со спецификацией управляемого размещения можно задавать спецификации базированного размещения, имеющие в частном случае вид или ***u*** или ***u*** **SET/В_УКАЗ(*p*)**, где *u* — имя базированной переменной, массива или структуры без явно заданного указателя (имена элементов базированных массивов или структур в данном случае не допустимы); *p* — имя переменной типа указатель. Если конструкция SET/В_УКАЗ(*p*) не задана, то начальный адрес отводимого участка памяти присваивается переменной *p′* типа указатель, заданной в описателе BASED/ОСНОВА(*p*') переменной (массива, структуры) для *u*. Если же конструкция SET/В_УКАЗ(*p*) задана, то начальный адрес отводимого участка памяти присваивается только переменной ***p*,** указанной в этой конструкции.

Участок памяти, отведенный посредством оператора ALLOCATE/ДАТЬ_ПАМЯТЬ под базированную переменную (массив, структуру), может быть освобожден в любой момент времени посредством оператора FREE/ВЕРНУТЬ_ПАМЯТЬ, рассмотренного в § 9.1. При этом должно быть указано имя базированной переменной (массива или структуры), идентичной той, которая использовалась при отведении памяти. Допустимы в данном случае имена с явно заданным указателем. Значение указателя, заданного явно или неявно, должно указывать на начало ранее отведенного участка памяти. В одном операторе FREE/ВЕРНУТЬ_ПАМЯТЬ могут быть указаны как базированные переменные (массивы, структуры), так и переменные (массивы, структуры) с управляемым размещением.

Одним из типичных примеров использования базированных переменных является работа со списками упорядоченных структур. Пусть, например, в процессе выполнения какой-либо программы, требуется хранить в оперативной памяти сведения о деталях (шифр детали, год начала выпуска и цена), упорядоченные по убыванию года начала выпуска, а при совпадающих годах — по возрастанию цены детали. Предположим также, что в процессе выполнения программы может потребоваться либо добавить в список, либо исключить из него сведения о какой- либо детали. Для обращения к элементам списка сведений о деталях опишем базированную структуру вида

```
опс 1 деталь основа(р),
      2 шифр текст(20),
      2 год десятичное(4),
      2 цена десятичное(10,2),
      2 ссылка указ;
```

В данной структуре элемент ссылка будет использован для организации связей внутри списка. Адрес 1-й структуры списка будем хранить в статической переменной типа указатель с внешним именем Р_детали. Пусть при пустом списке значение переменной Р_детали равно значению встроенной функции ПУСТО. В этом случае можно описать эту переменную оператором

```
опс Р_детали общее указ задать(пусто);
```

Перейдем теперь непосредственно к примерам обработки списка деталей.

Процедура, позволяющая вносить в список сведения о новой детали (в том числе и о первой детали) может иметь вид (здесь структура-параметр S содержит сведения о новой детали)

```
включение_детали_в_список: проц(s);
опс 1 деталь основа(р),
      2 шифр текст(20),
      2 год десятичное(4),
```

```
        2 цена десятичное(10,2),
        2 ссылка указ;
опс 1 s,
        2 n_шифр текст(20),
        2 n_год десятичное(4),
        2 n_цена десятичное(10, 2);
опс (P_детали общее, pn, px, p) указ;
//---- размещение новых сведений ----
дать_память деталь в_указ(pn);
pn-> шифр = n_шифр; pn-> год = n_год; pn-> цена = n_цена;
//---- включение новой структуры в список ----
p= P_детали; px = пусто;
цикл пока(p^ = пусто);
if n_год >год | (n_год = год & n_цена <цена) тогда
{;
   //---- добавление не в конец списка ----
   если px = пусто тогда P_детали = pn;
                        иначе px->ссылка = pn;
pn->ссылка = p;
возврат;
 };
px = p;
p = ссылка;
конец пока;
//---- добавление в конец списка ----
если px = пусто тогда P_детали = pn; иначе  px->ссылка = pn;
pn ->ссылка=пусто; // ссылка = null — признак конца списка
конец включение_детали_в_список;
```

Процедура, позволяющая по шифру детали найти адрес структуры, содержащей сведения об этой детали, может иметь вид (при отсутствии сведений о подобной детали возвращается «пустой» указатель).

```
найти_деталь: проц(шф) возвращает(указ);
опс 1 деталь основа(p),
        2 шифр текст(20),
        2 год десятичное(4),
        2 цена десятичное(10,2),
        2 ссылка указ
опс шф текст(20);
опс (P_детали общее, p) указ;
```

```
p = P_детали;
цикл пока(p^= пусто);
        если шф = шифр тогда возврат(p);
p=ссылка;
конец пока;
возврат(пусто);
конец найти_деталь;
```

Процедура, позволяющая по шифру детали исключить из списка соответствующие ей сведения, может иметь вид

```
исключение_детали_из_списка: проц(шф);
опс шф текст(20);
опс (P_детали общее, px, p) указ;
p = P_детали;
px = пусто;
цикл пока(p ^= пусто);
        если шф = шифр тогда
 {;
  //---- исключение структуры ----
   если px = пусто тогда P_детали = ссылка;
                          иначе px-> ссылка = ссылка;
   вернуть_память деталь;
   возврат;
 };
конец пока;
px = p;
p = ссылка;
end исключение_детали_из_списка;
```

Вывод на печать сведений обо всех деталях, содержащихся в списке может иметь вид

```
опс 1 деталь основа(p),
         2 шифр текст(20),
         2 год десятичное(4),
         2 цена десятичное(10,2),
         2 ссылка указ
опс (P_детали общее, p) указ;
p= P_детали;
цикл пока (p^ = пусто);
   писать с_новой в_виде(шифр, год, цена) (a(22), f(4), f(15,2));
   p=ссылка;
```

```
конец пока;
```

Проверь себя

1. Что является значением переменных типа указатель?
2. Какой описатель применяется при определении данных типа указатель? Какие операции применимы к этим данным?
3. Какой описатель применяется при описании базированных переменных, массивов и структур?
4. Могут ли имена базированных переменных быть внешними?
5. Каким образом определяется начало участка памяти, отведенного под базированную переменную, массив или структуру?
6. Что называется именем с явно заданным указателем?
7. Как определяется значение встроенной функции ADDR/АДРЕС?
8. Является ли синтаксически допустимым следующий фрагмент программы на языке PL/1?

```
опс x(100) бит(8), p ptr, b (1000)  бит(8) основа(p);
p= адрес(x);b =0;
```

9. Чему равно значение встроенной функции NULL/ПУСТО?
10. Зачем нужна встроенная функция NULL_PTR/ПУСТОЙ_УКАЗ?
11. Каковы будут значения указателей PA и PB после выполнения следующего фрагмента программы?

```
опс с текст(100) основа (ра);
дать_память с, с в_указ (рв);
```

11. Каким образом может быть освобождена память, отведенная под базированную переменную посредством оператора ALLOCATE/ ДАТЬ_ПАМЯТЬ?

10. ФАЙЛЫ, ВВОД И ВЫВОД

10.1. Понятие файла, файлы при потокоориентированном вводе и выводе

Вся информация, размещаемая на внешних запоминающих устройствах и на устройствах ввода или вывода, группируется в непересекающиеся совокупности, называемые *наборами данных***.** Так, например, набором данных может являться логически связанная совокупность сведений, хранимых на диске с логически связанными исходными данными или логически связанная совокупность сведений, выводимых на печать. Как правило, наборы данных разделяются на логические единицы, называемые *записями.* Размеры записей, так же как и размеры участков оперативной памяти, определяются в единицах информации, равных байту.

Программист, использующий язык PL/1, обращается к наборам данных посредством операторов ввода или вывода. При этом, однако, в программах на языке PL/1 никогда явно не указывается конкретные наборы данных, подлежащие обработке. Вместо этого в операторах языка PL/1 указываются абстрактные образы наборов данных, называемые *файлами.* Каждому файлу в программе приписывается имя, являющееся, вообще говоря, произвольным идентификатором. Имена файлов должны быть описаны явно — в операторах DECLARE/ОПИСАНИЕ. Вопросы описания имен файлов будут рассмотрены ниже.

Конкретные наборы данных, обрабатываемые программой на рассматриваемой версии языка PL/1, специфицируются файлами как понятиями операционной системы Windows. Файлы располагаются вне программы на языке PL/1, и именно поэтому одна и та же программа может обрабатывать различные файлы.

Любому обрабатываемому файлу в программе всегда сопоставляется некоторый файл операционной системы. Тем самым устанавливается связь между программно представляемым файлом и конкретным физическим набором данных, и благодаря этому любые действия над файлом в программе вызывают адекватные действия над физическим набором данных. В разные периоды выполнения программы одному и тому же программному файлу могут быть сопоставлены различные файлы, представленные во внешней среде. Процесс установки соответствия между файлом в программе и файлом в операционной системе называется *открытием файла*; процесс отмены подобного соответствия называется *закрытием файла***.** Операции открытия уже открытого файла приводят к возбуждению ошибки в программе, операции

закрытия закрытого файла не имеют смысла и игнорируются без сообщения об ошибке.

Явная установка соответствия между файлом в программе и файлом операционной системы осуществляется посредством специального оператора открытия файла, имеющего в простейшем случае вид **OPEN FILE/ОТКРЫТЬ ФАЙЛ (*f*);** где *f* — имя файла. При выполнении подобного оператора файл в программе связывается с файлом операционной системы, расположенным в текущей папке и имеющим также имя *f* и расширение .DAT. Например, после выполнения оператора открыть файл(INPUT); файл INPUT.DAT из текущей папки будет связан с именем INPUT и, следовательно, последующая обработка этого файла INPUT в программе вызовет обработку физического файла с именем INPUT и расширением .DAT. Посредством оператора OPEN/ОТКРЫТЬ указанного выше типа файл может быть связан лишь с одним и тем же (таким же) физическим файлом. Если же программисту требуется связывать файл в программе с различными файлами операционной системы, то он может воспользоваться оператором открытия вида

OPEN FILE (*f*) TITLE(*c*); или **ОТКРЫТЬ ФАЙЛ(*f*) ПО_ИМЕНИ(*c*);**

где *f* — имя открываемого файла; *c* — скалярное выражение, значение которого преобразуется в строку символов. Эта строка может содержать не только имя и расширение, но и путь к связываемому файлу *f*. Например, после выполнения оператора

открыть файл (IN) по_имени('C:\Users\Public\Documents\example.txt');

файл с именем IN в программе будет связан с файлом с именем 'C:\Users\Public\Documents\example.txt'.

С помощью одного оператора OPEN/ОТКРЫТЬ можно открыть несколько файлов. Оператор в этом случае будет иметь вид

OPEN/ОТКРЫТЬ d_1, d_2, ..., d_n;

где d_i (i=l, 2, . . . , n) спецификация открытия i-го файла, например

открыть файл(Fl), файл(F2) по_имени('A.LST'), файл(F3);

Программист не обязан явно открывать обрабатываемые им файлы. А именно, если оператор ввода или вывода обращается к закрытому файлу, то файл открывается автоматически. Но при этом он всегда связывается с физическим файлом, имеющим то же имя, что и имя файла, указанного в операторе ввода или вывода и расширением .DAT (отметим, что в рассмотренных в предыдущих главах операторах GET/ЧИТАТЬ и PUT/ПИСАТЬ имя файла явно не указывалось, однако при этом всегда подразумеваются файлы со стандартными именами, рассмотренными ниже).

Закрытие файла осуществляется посредством оператора, имеющего в простейшем случае вид

CLOSE FILE (*f*); или **ЗАКРЫТЬ ФАЙЛ (*f*);**

где *f* — имя файла. Допускаются составные операторы закрытия файлов вида

CLOSE/ЗАКРЫТЬ d_1, d_2, …,d_n;

где d_i (i=1, 2, ..., n) — спецификация закрытия i-го файла, например закрыть файл(FA), файл (FB); причем ключевой слово FILE/ФАЙЛ можно тоже опускать: закрыть (FA), (FB);

Отметим, что программист не обязан закрывать не требующиеся ему более файлы; операция закрытия таких файлов будет автоматически осуществлена операционной системой по окончании выполнения программы.

Перейдем к рассмотрению вопросов описания файлов. Имя любого файла должно быть явно описано в операторе DECLARE/ОПИСАНИЕ. Так же, как и большинству других имен, ему может быть приписан описатель EXTERNAL/ОБЩЕЕ (внешнее имя) или INTERNAL/МЕСТНОЕ (внутреннее имя); по умолчанию для имен файлов всегда предполагается описатель EXTERNAL/ОБЩЕЕ.

Помимо описателей EXTERNAL/ОБЩЕЕ и INTERNAL/МЕСТНОЕ все прочие описатели, допустимые для имен файлов, не допустимы ни для каких других имен и пишутся не в операторе DECLARE/ОПИСАНИЕ, а в операторе OPEN/ОТКРЫТЬ. В операторе же описания указывается лишь описатель FILE/ФАЙЛ; он указывает лишь, что данное имя является именем файла. Далее рассмотрим описатели файлов, обрабатываемых посредством операторов GET/ЧИТАТЬ и PUT/ПИСАТЬ, т.е. файлов, используемых при потокоориентированном вводе или выводе.

Описатель **INPUT/ДЛЯ_ВВОДА** указывает, что файл будет использоваться только для ввода, а описатель **OUTPUT/ДЛЯ_ВЫВОДА** — что файл будет использоваться только для вывода. Если ни один из этих описателей не задан, то подразумевается описатель INPUT/ДЛЯ_ВВОДА. Описатели INPUT/ДЛЯ_ВВОДА и OUTPUT/ДЛЯ_ВЫВОДА несовместимы друг с другом.

Описатель **PRINT/ДЛЯ_ПЕЧАТИ** указывает, что данные будут в конечном итоге выводиться на печать. При выводе списка данных для файлов с описателем PRINT/ДЛЯ_ПЕЧАТИ строки символов не заключаются в апострофы, а для прочих файлов — заключаются. Если задан описатель PRINT/ДЛЯ_ПЕЧАТИ, то по умолчанию предполагается описатель OUTPUT/ДЛЯ_ВЫВОДА.

Описатель **ENVIRONMENT/ДЛЯ_ОС** задает характеристики файлов, зависящие от конкретной реализации языка PL/1.

При потокоориентированном вводе и выводе данный описатель используется довольно редко и поэтому будет рассмотрен позднее.

Описатель **STREAM/ТЕКСТОВЫЙ** указывает, что файл будет использоваться для потокоориентированного ввода или вывода. Так как во всех допустимых случаях этот описатель подразумевается по умолчанию, то на практике он обычно не указывается. Все описатели файлов могут следовать в произвольном порядке.

Особый статус имеют имена стандартных файлов **SYSIN/СТД_ВВОД** и **SYSPRINT/СТД_ВЫВОД**. Эти имена не требуют явного описания. Файлу SYSIN/ СТД_ВВОД по умолчанию приписаны описатели

STREAM/ТЕКСТОВЫЙ,
INPUT/ДЛЯ_ВВОДА,
EXTERNAL/ОБЩЕЕ,

а файлу SYSPRINT/СТД_ВЫВОД —

STREAM/ТЕКСТОВЫЙ,
PRINT/ДЛЯ_ПЕЧАТИ,
EXTERNAL/ОБЩЕЕ,

Программист при желании может использовать эти имена для других целей, задав собственное явное описание. Отметим, что на практике использовать имена SYSIN/СТД_ВВОД и SYSPRINT/СТД_ВЫВОД не для обозначения стандартных файлов не рекомендуется, поскольку по умолчанию они связываются операционной системой с клавиатурой и экраном.

Характерной особенностью файлов является то, что описатели должны быть приписаны им не в операторе DECLARE/ОПИСАНИЕ, а при открытии файла, причем описатели, приписанные файлу при его открытии, игнорируются после закрытия файла.

Оператор открытия файлов имеет в общем случае вид (в том числе и при записеориентированном вводе или выводе):

OPEN/ОТКРЫТЬ $a_{11}\ a_{12}\ \ldots\ a_{1k_1}$, $n \geq 1$
$a_{21}\ a_{22}\ \ldots\ a_{2k_2}$, $k_i \geq 1$
…
$a_{n1}\ a_{n2}\ldots\ a_{nk_n}$;

где a_{ij} ($i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, k_i$) —либо любой описатель файла, кроме INTERNAL/МЕСТНОЕ, EXTERNAL/ОБЩЕЕ, FILE/ФАЙЛ, либо конструкция FILE/ФАЙЛ(f), где f — имя файла, либо конструкция TITLE/ПО_ИМЕНИ(c_i) (см. выше), либо рассмотренные ниже конструкции

LINESIZE/С_ДЛИНОЙ_СТРОКИ(m_i) и
PAGESIZE/С_ДЛИНОЙ_СТРАНИЦЫ(l_i).

В каждой последовательности a_{i1} a_{i2} … a_{ik_i}. должна быть одна конструкция FILE/ФАЙЛ(f_i) и не более чем по одной конструкции TITLE/С_ИМЕНЕМ(c_i), LINESIZE(m_i), PAGESIZE/(l_i) (или с эквивалентными русскими словами С_ДЛИНОЙ_СТРОКИ(m_i), С_ДЛИНОЙ_СТРАНИЦЫ(l_i)) .

Конструкция **LINESIZE/С_ДЛИНОЙ_СТРОКИ(*m*)**, где m — скалярное выражение с целочисленным значением, задается только для файлов, предназначенных для вывода. Она указывает, что длина выводимой строки должна быть равна m символам, например, после выполнения оператора открыть файл(SYSPRINT) для_печати с_длиной_строки(140);
строки, выводимые посредством файла SYSPRINT, будут состоять из 140 символов. Если конструкция с_длиной_строки не задана, то по умолчанию длина выводимой строки будет равна 80 символам. Следует учитывать, что максимальный размер печатаемой строки зависит не только от использования этой конструкции, но также и от параметров конкретного устройства печати и операционной системы.

Конструкция **PAGESIZE/С_ДЛИНОЙ_СТРАНИЦЫ(*l*)**, где l — скалярное выражение с целочисленным значением, задается только для файлов, которым приписан описатель PRINT/ДЛЯ_ПЕЧАТИ. Она указывает количество строк на стандартной странице. Вопросы перехода от одной страницы к другой при печати были рассмотрены в § 6.2 (см. также § 10.4). Если эта конструкция не задана, то по умолчанию количество строк в странице предполагается равным 60.

После закрытия файла конструкции С_ИМЕНЕМ, С_ДЛИНОЙ_СТРОКИ и С_ДЛИНОЙ_СТРАНИЦЫ, указанные при его открытии, игнорируются.

Приписывание файлам описателей производится также и при их автоматическом (неявном) открытии. При автоматическом открытии файла, вызванном выполнением оператора GET/ЧИТАТЬ, ему приписываются описатели STREAM/ТЕКСТОВЫЙ и INPUT/ДЛЯ_ВВОДА, а при автоматическом открытии файла, вызванном выполнением оператора PUT/ПИСАТЬ — описатели STREAM/ТЕКСТОВЫЙ и OUTPUT/ДЛЯ_ВЫВОДА.

Описатели, приписываемые при выполнении операторов записеориентированного ввода и вывода, будут рассмотрены в последующих параграфах.

В процедурах допускаются параметры типа файлов. С такими параметрами можно выполнять все те же операции, что и с обычными файлами. Аргумент, сопоставляемый подобному параметру, должен быть

открытым или закрытым файлом, возможно, также являющимся параметром. В описании параметра-файла в операторе DECLARE/ОПИСАНИЕ следует указывать описатель FILE/ФАЙЛ. Любые другие описатели в этом случае либо не допустимы. При обработке параметра-файла учитываются описатели конкретного аргумента, сопоставленного данному параметру. Однако при открытии параметров-файлов им могут быть приписаны любые описатели, не противоречащие описателям файла-аргумента.

В языке PL/1 допустимы также переменные со значениями типа файл. Имена таких переменных могут быть использованы во всех операторах ввода и вывода в том же контексте, что и имена файлов-констант. При этом будет обрабатываться тот файл, который является значением указанной переменной. Значение переменной типа файл устанавливается обычно посредством оператора присваивания, в правой части которого указывается либо имя файла-константы, либо имя переменной, либо имя процедуры-функции со значением типа файл. Начальное значение переменной типа файл может быть установлено посредством описателя INITIAL/ЗАДАТЬ. Переменные со значениями типа файл могут входить в состав массивов и структур. При описании переменных со значениями типа файл используются описатели FILE/ФАЙЛ и **VARIABLE/КОСВЕННОЕ**. Описатель VARIABLE/КОСВЕННОЕ может быть опущен, если переменная входит в состав массива или структуры, либо если ей приписан любой описатель, не допустимый для файлов-констант. Переменные со значениями типа файл могут быть параметрами процедур.

В языке PL/1 допустимы также процедуры-функции со значениями типа файл. В конструкциях и описателях RETURNS/ВОЗВРАЩАЕТ для подобных процедур должен быть задан описатель FILE/ФАЙЛ. Указатели таких процедур могут быть использованы во всех операторах ввода и вывода в том же контексте, что и имена файлов-констант.

К значениям типа файл применимы лишь две операции, а именно, операции сравнения = (равно) и ^= (не равно).

В заключение параграфа рассмотрим вопросы использования файлов в операторах потокоориентированного ввода (GET/ЧИТАТЬ) и вывода (PUT/ПИСАТЬ). В операторах данного типа, приведенных в предшествующих главах, нигде не указывалось имя файла. Это связано с тем, что в языке PL/1 имеется стандартное правило умолчания: если в операторе GET/ЧИТАТЬ не указано имя файла, то предполагается, что оператор обрабатывает файл с именем SYSIN/СТД_ВВОД, если же в операторе PUT/ПИСАТЬ не указано имя файла, то предполагается, что оператор обрабатывает файл с именем SYSPRINT/СТД_ВЫВОД. Обратите внимание на то, что в рассматриваемой

версии языка по умолчанию стандартный ввод связывается с клавиатурой, а стандартный вывод с экраном.

При желании программист, используя средства, рассмотренные выше, может открыть эти файлы, связав их с другими устройствами или физическими файлами, или, например, изменить длину строки стандартного вывода на число большее, чем 80.

При обработке с помощью операторов GET/ЧИТАТЬ или PUT/ПИСАТЬ файлов с именами отличными от стандартного ввода и вывода, в операторе должно быть указано имя файла в конструкции вида FILE/ФАЙЛ(*f*), где *f* — имя файла (или имя переменной или указатель функции со значением типа файл). Конструкция FILE/ФАЙЛ(*f*) может быть расположена в любом месте после ключевого слова GET/ЧИТАТЬ или PUT/ПИСАТЬ, но, естественно, до, а не внутри конструкции LIST, EDIT, DATA или SKIP (или внутри В_ВИДЕ, В_ФОРМЕ, С_ИМЕНАМИ или С_НОВОЙ).

В операторах, в которых опущено ключевое слово LIST/В_ВИДЕ, список данных должен следовать непосредственно за ключевым словом GET/ЧИТАТЬ или PUT/ПИСАТЬ, и поэтому никакие другие конструкции в таких операторах недопустимы.

Приведем примеры допустимых операторов потокоориентированного ввода и вывода (здесь A, B, C — имена переменных):

1) писать в_виде (A, B, C); задает вывод значений переменных A, B, C посредством файла СТД_ВЫВОД (по умолчанию на экран);
2) писать в_файл(t) с_именами(A); задает вывод значения переменной A посредством файла T; Ключевые слова ФАЙЛ, В_ФАЙЛ, ИЗ_ФАЙЛА эквивалентны;
3) писать с_новой в_файл(CC) в_форме(A, B) (ч(8)); задает вывод значений переменных A и B посредством файла CC, конструкция В_ФОРМЕ со списком форматов обязательна должна быть последней;
4) читать в _форме(A, B) (т(10)); задает ввод значений переменных A и B посредством файла СТД_ВВОД (по умолчанию с клавиатуры, конец ввода – комбинация клавиш CTRL+Z);

Приведем примеры недопустимых операторов потокоориентированного ввода и вывода:

1) писать стд_вывод в_виде(A, B); имя файла стд_вывод не записано в виде конструкции файл(стд_вывод);
2) писать в_виде файл(T) (A, B, C); конструкция файл расположена внутри конструкции в_виде;
3) читать в_форме(C) файл(IN) (ч(3)); конструкция файл расположена внутри конструкции в_форме;

4) читать файл в_виде(A, B); в конструкции файл опущено имя файла.

Рассмотрим содержательные примеры программ обработки файлов. Пусть необходимо вычислить сумму $\sum_{i=1}^{1000} x_i y_i$, где числа x_i и y_i размещены в текущей папке в отдельных файлах F1.DAT и F2.DAT.

Задача может быть решена посредством следующей программы

```
xy: проц главная;
опс (f1, f2) файл, (s, x,y) вещ, i точное;
s=0;
цикл i=1 до 1000;
  читать из_файла(f1) в_виде(x);
  читать из_файла(f2) в_виде(y);
  s +=x*y;
конец i;
писать с_новой с_именами(s);
конец xy;
```

Еще один пример. Требуется составить программу, которая-печатала бы сумму исходных чисел из каждого файла D1.TXT, D2.TXT, … D9.TXT. Искомая программа может иметь следующий вид:

```
печать_суммы: проц главная;
опс f файл, (n, j) точное, с текст(20) рд, (s, x) вещ;
//---- вводим с клавиатуры число файлов (до 9)
читать в_виде(n);
//---- цикл открытия и чтения файлов ----
цикл j = 1 до n;
        s = 0;
        когда конец_файла(f) идти файл_исчерпан;
        c= j; c=очистить(c); // убираем пробелы справа и слева
        открыть файл(f) для_ввода с_именем( 'd' ||c|| '.txt' );
  m:    читать из_файла(f) в_виде(x);
        s+=x;
        идти m;
  файл_исчерпан:
        закрыть файл(f);
        писать с_новой в_виде(s); // выдаем очередную сумму на экран
конец j;
конец печать_суммы;
```

В заключение рассмотрим две встроенные функции, используемые при работе с файлами, предназначенными для потокоориентированного ввода или

вывода. Встроенная функция **ONTERM** (без параметров) возвращает номер последнего считанного элемента в операторах ЧИТАТЬ С_ИМЕНАМИ или ЧИТАТЬ В_ВИДЕ. В случае если эти операторы содержат длинные списки элементов, бывает трудно определить, при чтении какого именно элемента из списка произошла ошибка. Значение встроенной функции **LINENO/ПЕЧАТАЕТСЯ_СТРОКА(*f*)**, где *f* — имя файла, которому приписан описатель PRINT/ДЛЯ_ПЕЧАТИ, равно номеру строки, заполняемой в данный момент при выводе посредством файла *f*. Тип значения функции ТОЧНОЕ(15). Если возникло состояние КОНЕЦ_СТРАНИЦЫ, то в этот момент номер строки на 1 больше, чем указано в конструкции С_ДЛИНОЙ_СТРАНИЦЫ при открытии файла *f*.

Проверь себя

1. Из каких единиц часто состоят наборы данных на внешних устройствах?
2. Что называется файлом?
3. Каким образом специфицируются наборы данных, которые будут обрабатываться программой на языке PL/1 при ее конкретном выполнении под управлением операционной системы?
4. Какие процессы называются открытием и закрытием файла?
5. Каким образом явно устанавливается соответствие между программным файлом и физическим набором данных?
6. Составьте пример оператора, открывающего несколько файлов.
7. В каких случаях производится автоматическое (неявное) открытие файла?
8. Посредством какого оператора осуществляется закрытие файлов (приведите примеры)?
9. Обязан ли программист задать закрытие для всех файлов, открытых в составленной им программе?
10. Перечислите описатели, допустимые для файлов, предназначенных для потокоориентированного ввода или вывода.
11. Какие особенности формирования выходных наборов данных для файлов с описателем PRINT/ДЛЯ_ПЕЧАТИ?
12. Укажите особенности имен стандартных файлов SYSIN/СТД_ВВОД и SYSPRINT/СТД_ВЫВОД.
13. На какой период связываются с файлом описатели, приписанные ему при его открытии?

14. Каковы синтаксис и семантика конструкций С_ДЛИНОЙ_СТРОКИ и С_ДЛИНОЙ_СТРАНИЦЫ задаваемых в операторе открытия файла?

15. Составьте оператор открытия файла, при котором размер печатаемой строки задавался бы равным 250 символам, а размер страницы — 50 строкам.

16. Каковы стандартные значения, предполагаемые для длины выводимой строки и размера выводимой страницы?

17. На какой период связываются с файлом конструкции С_ИМЕНЕМ, С_ДЛИНОЙ_СТРОКИ и С_ДЛИНОЙ_СТРАНИЦЫ, указанные при его открытии?

18. Укажите описатели, приписываемые файлом при автоматическом их открытии в процессе выполнения операторов PUT/ПИСАТЬ и GET/ЧИТАТЬ.

19. Перечислите основные особенности описания и использования параметров-файлов.

20. Перечислите основные особенности описания и использования в переменных и процедур-функций со значениями типа файл.

21. Укажите, какие файлы будут обрабатываться операторами ПИСАТЬ и ЧИТАТЬ, если в них имена файлов явно не указаны.

22. Посредством какой конструкции задается в операторе потокоориентированного ввода или вывода имя файла?

23. Составьте пример оператора потокоориентированного вывода, который бы задавал:

1) вывод значений переменных X и Y посредством файла стандартного вывода;

2) вывод значения переменной Z посредством файла с именем OUT.

24. Составьте пример оператора потокоориентированногэ ввода, который бы задавал:

1) ввод значений переменных X и Y посредством файла стандартного ввода, какая комбинация клавиш является признаком конца всего ввода?

2) ввод значений переменной Z посредством файла IN;

25. Укажите синтаксическую ошибку в следующих операторах:

1) читать стд_ввод в_виде(A, B);

2) писать с_именами файл(OUT) (X, Y);

3) писать файл в_виде(C);

26. Каковы значения встроенных функций ONTERM и ПЕЧАТАЕТСЯ_СТРОКА.

10.2. Записеориентированный ввод и вывод, файлы с последовательной организацией

При записеориентированном вводе программа на языке PL/1 тем или иным способом получает доступ к участку оперативной памяти, в которой без какого-либо автоматического преобразования передается определенная запись вводимого набора данных. Последующая обработка участка оперативной памяти, содержащего запись из набора данных, выполняется любыми доступными средствами языка PL/1 независимо от предшествующей операции записеориентированного ввода.

При записеориентированном выводе определенный участок оперативной памяти без какого-либо автоматического преобразования копируется на внешнее устройство, образуя новую или заменяя ранее созданную запись выходного набора данных. Заполнение информацией указанного участка оперативной памяти выполняется любыми доступными средствами языка PL/1 независимо от последующей операции записеориентированного вывода.

Записеориентированный ввод или вывод, по сравнению с потокоориентированным (по сути, текстом), позволяет программисту в большем объеме и более эффективно использовать разнообразные возможности работы с внешними устройствами. Можно утверждать, что в совокупности с остальными средствами языка PL/1 записеориентированный ввод и вывод обеспечивает не менее эффективное решение всех задач, которые могут быть решены с использованием потокоориентированного ввода и вывода (обратное утверждение при этом неверно). Однако если сравнить две программы (решающие одну и ту же задачу), первая из которых реализована с использованием потокоориентированного ввода и вывода, а вторая — записеориентированного, то, как правило, при составлении второй программы от программиста требуется больше усилий и больше знаний, нежели при составлении первой программы.

Операторы записеориентированного ввода и вывода, так же как и потокоориентированного, обращаются к наборам данных посредством файлов. Открытие и закрытие файлов осуществляется в данном случае так же, как это описано в § 10.1. Из описателей файлов, рассмотренных в § 10.1, к файлам, используемым при записеориентированном вводе и выводе, неприменимы описатели STREAM/ТЕКСТОВЫЙ и PRINT/ДЛЯ_ПЕЧАТИ. Однако к таким файлам применим ряд описателей, недопустимых при потокоориентированном вводе и выводе; причем все эти описатели должны быть заданы при открытии файла. Рассмотрим эти описатели.

Описатель **RECORD/НЕ_ТЕКСТОВЫЙ** указывает, что файл будет использоваться именно при записеориентированном вводе или выводе. Этот

описатель подразумевается по умолчанию, если задан любой из рассмотренных ниже описателей.

Описатель **UPDATE/ДЛЯ_ИЗМЕНЕНИЙ**, несовместимый с описателями INPUT/ДЛЯ_ВВОДА и OUTPUT/ДЛЯ_ВЫВОДА, указывает, что для обрабатываемого набора данных будут допустимы операции считывания записей и корректировки ранее созданных записей; новые записи при этом создаваться не будут. Подобные наборы данных могут быть размещены только на дисках.

Описатели **SEQUENTIAL/ПООЧЕРЕДНЫЙ** и **DIRECT/ПРЯМОЙ**, несовместимые друг с другом, задают способ доступа к набору данных. Последовательный доступ, задаваемый описателем SEQUENTIAL/ПОЧЕРЕДНЫЙ, будет рассмотрен позднее в данном параграфе. Прямой доступ, задаваемый описателем DIRECT/ПРЯМОЙ, будет рассмотрен в § 10.3, а также в § 13.7. Если не задан ни один из этих описателей, то по умолчанию предполагается описатель SEQUENTIAL/ПООЧЕРЕДНЫЙ.

Описатель **KEYED/ИНДЕКСНЫЙ** указывает, что в операторах ввода или вывода будет задаваться ключ записи. Наборы данных, в которых записям сопоставляются ключи, будут рассмотрены в § 10.3.

Простейшим и наиболее часто используемым вариантом организации наборов данных является последовательная организация. При такой организации записи в процессе вывода размещаются в наборе данных всегда в том порядке, в котором они выводятся, а в процессе ввода записи считываются всегда в том порядке, в котором они размещены — в наборе данных. Некоторые внешние устройства (например, печатающие) допускают лишь наборы данных с последовательной организацией. На дисках могут быть размещены как наборы данных с последовательной организацией, так и наборы с другими типами организации.

Описанные выше возможности считывания и формирования наборов данных с последовательной организацией называются последовательным доступом к наборам данных. Именно такой доступ задается для файла описателем SEQUENTIAL/ПООЧЕРЕДНЫЙ. Отметим, что такой доступ допустим и к наборам данных с другими типами организации; в то же время к наборам с последовательной организацией доступ возможен только последовательный.

Перейдем к рассмотрению операторов записеориентированного ввода и вывода, применяемых для наборов данных с последовательной организацией.

Формирование новой записи набора данных может быть выполнено с помощью оператора вывода записи:

WRITE FILE (*f*) FROM (*u*); или **ВЫВОД В_ФАЙЛ (*f*) ИЗ (*u*);**

где *f* — имя файла, связанного с набором данных (здесь и ниже *f* может быть также именем переменной или функции со значением типа файл); *u* — имя переменной, массива или структуры. Запись набора данных формируется из участка памяти, отведенного под переменную, массив или структуру *u*. Например, пусть описаны переменная A, массив B и структура C:

```
опс A вещ(53), B (10) текст(8), 1 C, 2(X, Y, Z) вещ;
```

Тогда после выполнения операторов

```
вывод в_файл (F1) из(A);
вывод из(B) в_файл (F2);
вывод в_файл(F3) из(C);
```

в файле F1 будет сформирована новая запись, состоящая из 8 байт, отведенных под значение переменной A; в файле F2 будет сформирована новая запись, состоящая из 80 байт, отведенных под массив B; в файле F3 будет сформирована новая запись, состоящая из 12 байт отведенных под структуру C. Обратите внимание, что конструкции FILE/ФАЙЛ и FROM/ИЗ могут следовать в любом порядке.

Однако не всегда удобно определять размер записи только по размеру записываемого элемента, поэтому в рассматриваемой версии языка допустима и конструкция вида

WRITE FILE (*f*) FROM (*u*, *n*); или ВЫВОД В_ФАЙЛ (*f*) ИЗ (*u*, *n*);

где *n* — явно указанный размер записи в байтах в виде выражения с целочисленным положительным значением. Если *n* окажется меньше размера записываемого элемента, — будет записана только часть элемента. Если *n* окажется больше размера записываемого элемента — после элемента будут дописаны байты с непредсказуемым содержимым. Если значение *n* окажется нулевым — размер записи будет определяться размером элемента и эта конструкция сведется к варианту без указания параметра *n*.

Переменная (массив, структура) *u*, задаваемая в конструкции FROM/ИЗ*(u)* оператора вывода записи, должна занимать связный участок оперативной памяти.

При формировании записи в случае, если *u* — переменная со строковым значением изменяющейся длины, будет возбуждено состояние ошибки. Обратите внимание, что если файл вывода имеет описатель STREAM/ТЕКСТОВЫЙ, то конструкция вывод в_файл(f) из(u); становится допустимой и в этом случае выводится только участок памяти, занимаемый текущим значением переменной.

Для ввода записи применяется оператор чтения записи, имеющий две различных формы. Первая форма оператора имеет вид:

READ FILE (*f*) INTO (*u*); или **ВВОД ИЗ_ФАЙЛА (*f*) В_ПЕРЕМ(*u*);**

где *f* — имя входного файла; *u* — имя переменной, массива или структуры. При выполнении такого оператора очередная запись набора данных, соответствующего файлу *f*, помещается в участок оперативной памяти, отведенный под *u*. Напоминаем, что русские ключевые слова ФАЙЛ, В_ФАЙЛ, ИЗ_ФАЙЛА эквивалентны. Например, пусть имеется оператор описания

```
опс A вещ(53), B(10) текст(8), 1 C, 2(X, Y, Z) вещ;
```

Тогда после выполнения операторов

```
ввод из_файла(F1) в_перем (A);
ввод из_файла(F2) в_перем (B);
ввод в_перем (C) из_файла (F3);
```

в участок памяти, отведенный под переменную A, будет считана очередная запись файла F1; в участок памяти, отведенный под массив B, будет считана очередная запись файла F2; в участок памяти, отведенный под структуру C, будет считана очередная запись файла F3. Обратите внимание на то, что конструкции FILE/ИЗ_ФАЙЛА и INTO/В_ПЕРЕМ могут следовать в любом порядке.

Как и в предыдущей конструкции не всегда удобно определять размер записи только по размеру вводимого элемента, поэтому в рассматриваемой версии языка допустима и конструкция вида

READ FILE (*f*) INTO(*u, n*); или **ВВОД ИЗ_ФАЙЛА(*f*) В_ПЕРЕМ (*u, n*);**

где *n* — явно указанный размер читаемой записи в байтах в виде выражения с целочисленным положительным значением. Если *n* окажется меньше размера записываемого элемента, — будет введена только часть элемента. Если *n* окажется больше размера вводимого элемента — произойдет ошибка с непредсказуемыми последствиями. Если значение *n* окажется нулевым — размер читаемой записи будет определяться размером элемента и эта конструкция сведется к варианту без указания параметра *n*.

Переменная (массив, структура) *u*, указываемая в рассмотренной форме оператора READ/ВВОД должна отвечать тем же требованиям, что и переменная (массив, структура) *u*, указываемая в операторе WRITE/ВЫВОД (см. выше).

В случае если *u* — переменная со строковым значением изменяющейся длины, то будет возбуждено состояние ошибки. Обратите внимание, что если

файл вывода имеет описатель STREAM/ТЕКСТОВЫЙ, то конструкция ввод из_файла(*f*) в_перем(*u*); становится допустимой и в этом случае вводится только участок памяти, занимаемый текущим строкой файла *f*.

Вторая форма оператора чтения записи имеет вид

READ FILE (*f*) SET (*p*); или **ВВОД ИЗ_ФАЙЛА (*f*) В_УКАЗ (*p*);**

где *f* — имя входного файла; *p* — имя переменной типа указатель. При выполнении подобного оператора адрес участка памяти внутри автоматически выделенного буфера, в который считывается очередная запись файла *f*, помещается в указатель *p*. После этого программист, используя указатель *p*, может получить доступ к информации, содержащейся во введенной записи, посредством любой базированной переменной (массива, структуры). Например, если описана базированная структура

```
опс 1 S основа(P), 2(M точное(31), X вещ(53));
```

то после выполнения оператора

```
ввод из_файла(F) в_указ(P);
```

значение элемента M структуры S будет представлено первыми четырьмя байтами очередной записи файла F, а значение элемента X будет представлено следующими восемью байтами этой же записи. Конструкции FILE/ИЗ_ФАЙЛА и SET/В_УКАЗ в рассмотренной форме оператора READ/ВВОД могут следовать в любом порядке.

В файлах-наборах данных, размещенных на дисках, можно заменять ранее созданные записи. При этом соответствующему файлу должен быть приписан описатель **UPDATE/ДЛЯ_ИЗМЕНЕНИЙ**. При последовательном доступе возможна замена только последней записи, считанной оператором чтения записи. Замена осуществляется с помощью оператора вида

REWRITE FILE (*f*); или **ПЕРЕЗАПИСАТЬ ФАЙЛ(*f*);**

где *f* — имя обрабатываемого файла.

Оператор замены записи применим только в том случае, когда последняя запись файла *f* была введена посредством оператора чтения записи с конструкцией SET/В_УКАЗ. При этом программист должен сформировать новую запись в пределах того же участка буфера, куда была введена старая запись. Например, после выполнения последовательности операторов

```
опс F файл, A основа(P) текст(1), P указ;
открыть файл(F) для_исправлений;
ввод из_файла(F) в указ(P);
A = 'X';
перезаписать файл(F);
```

первый символ первой записи файла F будет заменен на символ X; остальная часть записи останется без изменений.

В тех случаях, когда рассмотренные выше операторы записеориентированного ввода или вывода применяются к закрытому файлу, то файл автоматически открывается с приписыванием ему описателя RECORD/НЕ_ТЕКСТОВЫЙ. Кроме того: при выполнении оператора WRITE/ВЫВОД файлу автоматически приписывается описатель OUTPUT/ДЛЯ_ВЫВОДА (при условии, что файлу не приписан явно описатель UPDATE/ДЛЯ_ИЗМЕНЕНИЙ); при выполнении оператора READ/ВВОД — описатель INPUT/ДЛЯ_ВВОДА (если явно не задан описатель UPDATE/ДЛЯ_ИСПРАВЛЕНИЙ); при выполнении оператора REWRITE/ПЕРЕЗАПИСАТЬ — описатель UPDATE/ДЛЯ_ИСПРАВЛЕНИЙ.

При создании набора данных, включая наборы с последовательной организацией, программист обязан задать количественные и качественные характеристики формата блоков и записей набора. Эти сведения могут быть указаны в описателе **ENVIRONMENT/ДЛЯ_ОС** файла в программе, сопоставляемого физическому набору данных. Этот описатель имеет в общем случае вид

ENVIRONMENT(p_1, p_2, …, p_n) или ДЛЯ_ОС(p_1, p_2, …, p_n) ($n \geq 1$),

где p_i (i=1, 2, ..., n) — параметр описателя, задающий различные сведения о физическом файле или наборе данных, ориентированные на конкретную реализацию языка PL/1 и зависящие не от языка, а от операционной системы. Параметры не могут следовать в любом порядке. Для слова ENVIRONMENT допустимо сокращение **ENV**.

Для рассматриваемой версии языка могут быть заданы следующие параметры:

A - для текстовых файлов при вводе этот параметр («A» - от названия ANSII) имеет смысл только для оператора чтения READ/ВВОД в переменную со строковым значением переменной длины. При задании этого параметра, символ "^" в читаемом файле воспринимается, как и в строчных константах данной версии PL/1, т.е. гасит три старших разряда в следующим за ним символе, превращая его в управляющий.

Таким образом, ^I превращается в символ табуляции, ^M - в символ конца строки и т.д. В текстовом файле становится возможным записывать управляющие символы и другую информацию (например, атрибуты цветов символов), не используя непечатных символов. Сам символ "^" записывается как два подряд ("^^").

Если задана опция A для файлов для вывода («A» - от слова *append*) и такой файл уже существует, строки дописываются в конец существующего файла. Параметр A, если он есть, должен стоять первым.

F(*i*) – этот параметр определяет файл с записями фиксированной длины, размером *i* байт каждая, выражение *i* должно быть типа точное(31). По умолчанию *i* равно 128 байтам. Размер записи может быть любой длины от 1 до 2**31-1.

B(*i*) - этот параметр определяет размер буфера ввода/вывода (т.е. внутреннего участка памяти, куда вначале будут попадать считанные записи, или куда сначала будут переписываться записи для вывода) размером *i* байт, выражение *i* - должно быть типа точное(31). Физически обмен с файлом будет идти порциями по *i* байт. По умолчанию *i* равно 128 байтам. Если значение *i* равно -1 – файл «отображается на память» средствами Windows (см. § 13.7).

Для повышения скорости обмена параметры и F и B выгодно делать кратными 512 байтам, поскольку физический обмен с дисками часто идет секторами такой длины.

Если есть параметр F, должен быть описатель KEYED/ИНДЕКСНЫЙ, если есть B, но нет F, то считается, что записи переменной длины и описателя KEYED/ ИНДЕКСНЫЙ не может быть. Если есть и параметр F и параметр B, F - должен стоять первым. Ключевое слово ENVIRONMENT также можно заменять словом **OPTIONS.**

При решении вопроса о выборе формата для создаваемого набора данных программист в первую очередь должен учитывать ограничения на применение тех или иных форматов в каждом конкретном случае. Желательно стремиться к тому, чтобы размеры записей и буферов приводили к минимальному числу обращений к физическим устройствам.

Проверь себя

1. Приведите краткую характеристику записеориентированного ввода и вывода.
2. Перечислите описатели, допустимые только для файлов, используемых при записеориентированном вводе или выводе.
3. В каких случаях файлу приписывается описатель UPDATE/ ДЛЯ_ИЗМЕНЕНИЙ?
4. Что называется последовательным доступом к файлам как к наборам данных?
5. Укажите основные особенности последовательной организации файлов как наборов данных.
6. Из каких данных будут сформированы очередные записи файлов Fl, F2 и F3 после выполнения операторов

```
опс X вещ(53), Y(10) текст(8), 1 Z, 2(A, B, C) вещ; X, Y, Z = 5;
вывод в_файл(Fl) из(X);
```

```
вывод из(Y) в_файл(F2);
вывод в_файл(F3) из(Z);
```

7. В каких случаях число байт при вводе и выводе будет не равно размеру вводимых и выводимых элементов?

8. Каковы требования к переменной, массиву или структуре ***v,*** задаваемой в конструкции FROM/ИЗ оператора вывода записи?

9. В чем состоит различие во вводе и выводе строк изменяющейся длины операторами ВВОД и ВЫВОД от всех других типов переменных?

11. Перечислите формы оператора чтения записи, применимые при обработке наборов данные с последовательной организацией.

12. Каким требованиям должны удовлетворять переменная, массив или структура, указанные в конструкции INTO/В_ПЕРЕМ оператора чтения записи?

13. Что будет напечатано после выполнения фрагмента программы:

```
опс X текст(3) основа(P);
открыть файл(T) не_текстовый для_ввода;
цикл i=1 до 3;
ввод из_файла (T) в_указ(P);
писать в_виде(X) (т);
конец i;
```

14. Какой описатель всегда явно или неявно приписывается файлу, к которому применяется оператор замены записи?

15. Какие описатели приписываются файлу при его неявном открытии во время выполнения каждого из операторов записеориентированного ввода или вывода, рассмотренного в данном параграфе?

16. Укажите назначение всех параметров описателя ENVIRONMENT/ ДЛЯ_ОС, применяемых для задания размеров и формата записей.

17. Пусть входной поток, соответствующий файлу SYSIN, представляет собой последовательность строк по 80 символов каждая. Требуется напечатать только те их этих строк, которые содержатся во входном потоке, по крайней мере, в 3 экземплярах. При этом считать, что в оперативной памяти не могут быть одновременно размещены все строки входного потока.

10.3. Прямой доступ к наборам данных

Прямым доступом называется такой доступ к наборам данных, при котором независимо от ранее обработанной записи в очередной операции ввода или вывода может быть считана или изменена любая запись набора данных. Использование прямого доступа задается описателем **DIRECT/ ПРЯМОЙ** соответствующего файла. Прямой доступ возможен только к наборам данных, размещенным на дисках.

В данном параграфе будет рассмотрена простейшая из организаций файлов-наборов данных, допускающих прямой доступ (напомним, что при последовательной организации наборов к ним возможен лишь последовательный доступ), при этой организации все записи файла последовательно нумеруются целыми числами от 0 до N-1, где N — общее количество записей в файле. Нумерация записей производится в процессе создания данных, память под номер записи в файле на диске фактически не отводится.

На использование таких наборов данных имеется следующее ограничение: в таком наборе допустимы лишь записи фиксированной длины.

Создание наборов данных с такой организацией можно осуществлять посредством операторов вида

WRITE FILE (*f*) FROM (*u*) KEYFROM(*l*); или

ВЫВОД В_ФАЙЛ(*f*) ИЗ(*u*) ИНДЕКС(*l*);

В рассматриваемой версии языка ключевые слова KEY/ИНДЕКС и KEYFROM/ИЗ_ИНДЕКСА эквивалентны.

В данном случае номера записей *l*, указываемые в конструкции KEYFROM/KEY/ИНДЕКС/ИЗ_ИНДЕКСА, могут следовать в любом порядке. При этом «пропущенные» по порядку номеров записи тоже создаются, но остаются пустыми. В целом, работа с файлами-наборами прямого доступа напоминает работу с массивами или массивами структур, когда возможно обращение к произвольному элемента массива по индексу. Можно считать, что в данном случае это тот же массив или массив структур, но он располагается в файле на диске, а не в памяти.

Обращение к созданному набору данных с организацией прямого доступа может осуществляться посредством как последовательного, так и собственно прямого доступа. При последовательном доступе используются те же формы операторов чтения и замены записей, что и для наборов данных с последовательной организацией. Дополнительно в операторах чтения записи может быть использована конструкция **KEYTO/ДЛЯ_ИНДЕКСА*(u)***, где *u* — переменная со значением типа точное(31); при выполнении оператора этой

переменной присваивается значение, равное номеру записи. Конструкции KEYTO/ДЛЯ_ИНДЕКСА, FILE/ФАЙЛ, INTO/В_ПЕРЕМ и SET/В_УКАЗ могут следовать в любом порядке. Напомним, что при использовании конструкции KEYTO/ДЛЯ_ИНДЕКСА файлу должен быть приписан описатель KEYED/ ИНДЕКСНЫЙ.

Для считывания записей при прямом доступе используются операторы чтения записи вида

READ FILE (*f*) INTO (*u*) KEY(*e*); или

ВВОД ИЗ_ФАЙЛА (*f*) В_ПЕРЕМ (*u*) ИНДЕКС(*e*);

где *e* — целочисленное выражение, задающее номер записи по тем же правилам, что и в конструкции KEYFROM/ИЗ_ИНДЕКСА (см. выше); синтаксис и семантика конструкций FILE/ФАЙЛ и INTO/В_ПЕРЕМ те же, что и для операторов чтения записи, рассмотренных в § 10.2. Разумеется, явно задавать длину считываемого здесь нельзя.

В данном случае оператор выполняется так же, как и при последовательном доступе при наличии конструкции INTO/В_ПЕРЕМ с той разницей, что считывается именно та запись, номер которой указан в конструкции KEY/ИНДЕКС. Порядок конструкций KEY/ИНДЕКС, FILE/ФАЙЛ и INTO/В_ПЕРЕМ может быть произвольным. Например, после выполнения операторов

```
ввод из_файла (F) в_перем (T1) индекс (543);
ввод из_файла (F) индекс (0) в_перем(T2);
ввод из_файла (F) в_перем (T3) индекс (C);
```

значение переменной Tl будет задано записью с номером 543, значение переменной T2 будет задано записью с номером 0, значение переменной T3 будет задано записью с номером, равным значению переменной C. Отметим, что при считывании записей из файла прямого доступа посредством и последовательного, и прямого доступа, передаются как записи, включенные в набор программистом, так и пустые записи.

Для замены записи в наборе данных с прямым доступом используется оператор записи, конструкция REWRITE/ПЕРЕЗАПИСАТЬ не применяется.

Проверь себя

1. Какой доступ к наборам данных называется прямым?
2. Каким описателем файлов задается прямой доступ?
3. Укажите основные особенности структуры файлов-наборов данных с прямым доступом.
4. В каких случаях файлу должен быть приписан описатель KEYED/ИНДЕКСНЫЙ?

5. Какое значение присваивается переменной *u*, если конструкция КЕУТО/ДЛЯ_ИНДЕКСА(*u*) указана в операторе последовательного считывания записи?

6. Опишите синтаксис и семантику операторов чтения и записи, допустимых при прямом доступе.

10.4. Обработка прерываний при вводе и выводе

В языке PL/1 выделен ряд состояний, возникающих в процессе ввода или вывода, для которых программист может задать собственную обработку прерываний. Имена всех состояний ввода-вывода имеют вид *c(f)*, где *c* — идентификатор, задающий общее имя (тип) состояния; *f* — имя файла (может быть также именем переменной или процедуры-функции со значением типа файл). Примером имени состояния ввода-вывода является использовавшееся ранее в операторах обработки прерываний имя ENDFILE(SYSIN) или КОНЕЦ_ФАЙЛА(СТД_ВВОД), где ENDFILE/КОНЕЦ_ФАЙЛА общее имя состояния, а конкретное имя файла – стандартный ввод. Рассмотрим подробно каждое состояние ввода-вывода.

Состояние **ENDFILE/КОНЕЦ_ФАЙЛА** может возникнуть либо при выполнении оператора потокоориентированного ввода GET/ЧИТАТЬ, либо при выполнении оператора чтения записи READ/ВВОД при последовательном доступе. Но также это состояние может возникнуть и при записи, если уже исчерпано свободное место на диске. При потокоориентированном вводе состояние ENDFILE/КОНЕЦ_ФАЙЛА возникает в том случае, если при попытке начать ввод очередного элемента данных будет обнаружен конец входного потока (символ с кодом '1А'Б4). Если же конец входного потока будет обнаружен в процессе ввода очередного элемента данных, состояние ENDFILE/КОНЕЦ_ФАЙЛА будет иметь другой «субкод» состояния (код самого состояния в данной версии равен 5):

КОНЕЦ ФАЙЛА(1)	при чтении строки переменной длины
КОНЕЦ ФАЙЛА(2)	не получилась запись байта
КОНЕЦ ФАЙЛА(3)	при выполнении read/ввод
КОНЕЦ ФАЙЛА(4)	не получился оператор write/вывод
КОНЕЦ ФАЙЛА(5)	не получилась запись skip/с_новой
КОНЕЦ ФАЙЛА(7)	при чтении по заданному формату

При записеориентированном вводе состояние ENDFILE/КОНЕЦ_ФАЙЛА возникает при попытке выполнить оператор чтения записи после того, как считана последняя запись набора данных.

Стандартная обработка прерывания для данного состояния заключается в печати диагностического сообщения и создании состояния ERROR/ОШИБКА.

При нормальном возврате после обработки прерывания для состояния ENDFILE/КОНЕЦ_ФАЙЛА выполнение программы продолжается с оператора, следующего за прерванным.

Состояние **ENDPAGE/КОНЕЦ_СТРАНИЦЫ** возникает только при потокоориентированном выводе для файлов с описателем PRINT/ДЛЯ_ПЕЧАТИ (т.е. в случае, когда данные предназначены для печати). Возникновение состояния является признаком того, что предыдущая выводимая страница заполнена полностью и сделана попытка начать вывод на новую страницу. Состояние может возникнуть либо при передаче данных, либо при выполнении действий, задаваемых форматами (конструкциями) LINE/ПЕРЕВОД_СТРОКИ или SKIP/С_НОВОЙ (см. гл. 6). Если нет обработки прерывания для этого состояния, работа программы продолжается со следующего оператора. На практике обработка прерывания для состояния ENDPAGE/КОНЕЦ_СТРАНИЦЫ обычно включает в себя переход к новой странице; напомним, что он задается форматом (конструкцией) PAGE/ПЕРЕВОД_СТРАНИЦЫ в списке форматов в операторе потокоориентированного вывода (PUT/ПИСАТЬ).

Однако если состояние было вызвано оператором SIGNAL/СИГНАЛ, то стандартная обработка прерывания не предусматривает в этом случае каких-либо действий.

При нормальном возврате после обработки прерывания продолжается прерванная передача данных. Если же состояние ENDPAGE/КОНЕЦ_СТРАНИЦЫ возникло при выполнении действий, заданных конструкциями (форматами) LINE/ПЕРЕВОД_СТРОКИ или SKIP/С_НОВОЙ, то после обработки прерывания эти действия не возобновляются.

Рассмотрим пример. Пусть какой-либо длинный текст необходимо распечатать с делением на страницы таким образом, чтобы в 1-й строке каждой страницы помещался бы ее номер, а сам текст выводился бы со 2-й строки каждой страницы. Задача может быть решена посредством выполнения операторов

```
когда конец_страницы (стд_вывод)
блок;
        писать в_форме(n, с_новой) (перевод_страницы, п(58), ч(3));
        n += 1;
конец;
n=1;
сигнал конец_страницы (стд_ввод);
```

```
// начало вывода текста
```

Состояние **KEY/ИНДЕКС** возникает при записеориентированном вводе или выводе для файлов, которым явно или по умолчанию приписан описатель KEYED/ИНДЕКСНЫЙ. При работе с прямым доступом это состояние возникает в следующих случаях:

— значение выражения в конструкции KEYFROM/ИЗ_ИНДЕКСА или KEY/ИНДЕКС не задает должным образом номер записи;

— нет места на диске для создания записи с указанным номером;

— запись с указанным номером в файле отсутствует.

Стандартная обработка прерывания для состояния KEY/ИНДЕКС заключается в печати диагностического сообщения и создании состояния ошибки.

При нормальном возврате после обработки прерывания для состояния KEY/ИНДЕКС выполнения программы продолжается с оператора, следующего за прерванным.

Рассмотрим пример. Пусть программисту требуется установить количество записей в файле прямого доступа. Тогда после выполнения операторов

```
открыть файл(f) для_ввода прямой индексный;
когда индекс (f) идти out;
цикл i = 0 с_шагом 1;
ввод из_файла(f) в_перем(c) индекс(i);
конец i;
out:;
```

значение переменной *i* будет равно количеству записей в файле *f*.

Состояние **UNDEFINEDFILE/НЕТ_ФАЙЛА** возникает при безуспешной попытке открыть какой-либо файл. Наиболее часто данное состояние возникает в следующих двух случаях:

— отсутствует файл с требуемым при открытии именем;

— файл в данный момент монопольно занят другой задачей.

Стандартная обработка прерывания для этого состояния заключается в печати диагностического сообщения и создании состояния ошибки.

Если состояние возникло во время неявного (автоматического) открытия файла, то при нормальном возврате после обработки прерывания делается попытка продолжить выполнение прерванного оператора; при этом если во время обработки прерывания файл не был должным образом открыт, то создается состояние ошибки. Если состояние возникло во время выполнения оператора открытия файла, то при нормальном возврате после обработки прерывания выполняется следующий оператор программы (если в операторе

задано открытие нескольких файлов, то состояние UNDEFINEDFILE/НЕТ_ФАЙЛА может возникнуть после попыток открыть каждый файл); при этом если не удалось открыть несколько файлов, то состояние возникнет для каждого неоткрытого файла в той последовательности, в которой файлы указаны в операторе открытия).

Для ключевого слова UNDEFINEDFILE допустимо сокращение **UNDF**.

Рассмотрим пример. Пусть имеется несколько файлов данных с именами T*nnn*.TXT, где *nnn* — трехзначное десятичное число; причем числа *nnn* занимают весь диапазон целых чисел от 101 до N, где N-100 равно количеству этих входных файлов. Требуется для каждого входного файла распечатать сумму представленных в нем чисел, при этом количество наборов данных может быть произвольным и явно программе не передается. Задача может быть решена с помощью следующей программы:

```
psn: проц главная;
опс f файл, i точное, (x, s) вещ, c текст(20) рд;
когда нет_файла(f) идти out;
цикл i = 101 с_шагом 1;
        когда конец_файла(f) идти конец_ц;
        c=i; c=очистить(c);
        открыть файл(f) для_ввода с_именем('t'||c||'.txt');
        s = 0;
        m: читать из_файла(f) в_виде(x);
        s += x;
        идти m;
        конец_ц: закрыть файл(f);
        писать с_новой в_виде(i, s);
 out: конец i;
конец psn;
```

При возникновении любого состояния ввода-вывода значение встроенной функции **ONFILE/КОГДА_ФАЙЛ**, не имеющей параметров, устанавливается равным строке символов, изображающей имя файла, при обработке которого произошло прерывание. В иных ситуациях значение функции ONFILE/КОГДА_ФАЙЛ равно пустой строке символов.

При возникновении состояний KEY/ИНДЕКС в процессе работы с файлом, которому приписан описатель KEYED/ИНДЕКСНЫЙ, значение встроенной функции **ONKEY/КОГДА_ИНДЕКС** устанавливается равным номеру записи, при обработке которой произошло прерывание. В иных ситуациях значение функции ONKEY/КОГДА_ИНДЕКС равно нулю.

Проверь себя

1. Каков общий вид имен состояний ввода-вывода?
2. В каких случаях возникает состояние ENDFILE/КОНЕЦ_ФАЙЛА?
3. В чем заключается стандартная обработка прерываний для состояния ENDFILE/КОНЕЦ_ФАЙЛА?
4. В каких случаях возникает состояние ENDPAGE/КОНЕЦ_СТРАНИЦЫ, в чем заключается стандартная обработка прерываний для этого состояния (в том числе и в случае состояний, вызванных оператором SIGNAL/СИГНАЛ)?
5. Как продолжается выполнение программы при нормальном возврате после обработки прерывания для состояния ENDPAGE/КОНЕЦ_СТРАНИЦЫ?
6. Пусть в файле имеется некоторый текст. Составьте программу, которая вводит весь этот текст и печатает его по страницам строками размером по 80 символов. При этом в 1-й строке каждой страницы должен быть напечатан ее номер, а текст должен размещаться со 2-й строки (указание: при решении задачи используйте обработку состояний КОНЕЦ_СТРАНИЦЫ).
7. В каких случаях возникает состояние KEY/ИНДЕКС?
8. Приведите примеры ситуаций, при которых возникает состояние UNDEFINEDFILE/НЕТ_ФАЙЛА.
11. Укажите, каковы значения встроенных функций ONFILE/КОГДА_ФАЙЛ и ONKEY/КОГДА_ИНДЕКС в тех или иных ситуациях.

11. ПОДГОТОВКА ПРОГРАММ К ИСПОЛНЕНИЮ

11.1. Понятие программно-аппаратной среды и ее влияние на язык

Программы выполняются на аппаратных средствах, при этом сами средства работают под управлением специальной сервисной программы — *операционной системы*. Совокупность используемых аппаратных средств и операционной системы составляют программно-аппаратную среду исполнения программы.

Языки программирования (включая и PL/1) проектируются так, чтобы программы получались, как можно менее зависимы от конкретной программно-аппаратной среды и ее особенностей. Но сделать программы полностью независимыми от среды практически невозможно, да это и не нужно, поскольку программы в ряде случаев как раз и должны использовать особенности среды для эффективного достижения конечного результата.

Описываемая версия языка предназначена для работы в программно-аппаратной среде 64-х разрядной операционной системы Windows с архитектурой процессоров x86-64. Это определяет следующие особенности реализации языка PL/1 и компилятора:

— возможна адресация до 2^{64} байт, при этом длина переменных типа указатель составляет 8 байт;

— целочисленные переменные могут изменять свое значение в диапазоне от -2^{63} до $2^{63}-1$, при этом их длина составляет 8 байт;

— битово-строчные переменные могут быть длиной до 64 бит, т.е. также до 8 байт длиной;

— возможно одновременное исполнение нескольких процедур PL/1 или даже одновременное исполнение нескольких экземпляров одной процедуры, поскольку чаще всего имеется несколько физических процессоров или их частей-«ядер», работающих одновременно.

Заметим, что не запрещено использовать и переменные меньшей разрядности (кроме указателей), занимающих, например, 1, 2 или 4 байта. Таким образом, речь идет только о предельных значениях переменных. Например, в 32-х разрядной программно-аппаратной среде указатели занимали бы 4 байта и могли адресовать до 2^{32} байт, а целочисленные переменные, также занимая 4 байта, могли менять свое значение в диапазоне от -2^{32} до $2^{32}-1$.

Существуют и другие особенности реализации языка, связанные со средой. Так, русскоязычному программисту иногда приходится работать сразу с двумя наборами русских букв, получивших названия «кириллица DOS» и

«кириллица Windows», что было связано с необходимостью обратной совместимости со средой MS DOS. Хотя есть стандартные процедуры перевода из одного набора в другой, это неудобно при задании текстовых констант. Поэтому в рассматриваемой версии языка добавлен квалификатор W текстовых констант, который ставится после закрывающего апострофа или после повторителя (если повторитель был задан). Например

```
опс S1 текст(*) рд постоянное задать('Привет!');
опс S2 текст(*) рд постоянное задать('Привет!'W);
```

здесь первый символ символьно-строчной S1 имеет код '8F'Б4 (кириллица DOS), а первый символ S2 имеет код 'CF'Б4 (кириллица Windows).

Зависимость от программно-аппаратной среды в языке PL/1 в некотором смысле обособлена и в текстах программ. Во-первых, это описатель ENVIRONMENT/ДЛЯ_ОС в операторе открытия файлов. Сюда же нужно отнести и описатель TITLE/С_ИМЕНЕМ, поскольку «внешнее» имя файла не входит в понятия языка. Во-вторых, это встроенная функция и псевдопеременная UNSPEC/МАШ_КОД, которая явно зависит от конкретной аппаратуры. В-третьих, это определяемые переменные DEFINED/НА_МЕСТЕ, поскольку требуется знать внутреннее представление данных. Наконец, описатель процедуры MAIN/ГЛАВНАЯ в исходном варианте языка помещался внутрь еще одного описателя - OPTIONS, т.е. первоначально заголовки главных процедур выглядели как M: PROC OPTIONS(MAIN); хотя понятие «главная» — конечно, входит в понятия языка, а не только операционной системы. В рассматриваемой версии языка описатель заголовка процедуры OPTIONS стал необязателен, но может быть использован для задания нестандартного начального размера стека, например

```
M:PROC OPTIONS(MAIN, STACK(10000));
```

Напомним, что ключевые слова ENVIRONMENT, OPTIONS и ДЛЯ_ОС эквивалентны, поскольку все они предназначены для сообщения некоторой информации операционной системе.

Проверь себя

1. Что такое программно-аппаратная среда?
2. Как связан диапазон допустимых адресов в переменных типа указатель с разрядностью среды?
3. Зачем в рассматриваемую версию языка введен дополнительно квалификатор W для текстовых констант?
4. Как найти в тексте программы на языке PL/1 части, которые могут зависеть от программно-аппаратной среды?

11.2. Методы и стили программирования. Императивный метод

Правильного результата при выполнении программы можно добиться при организации этой программы по разным совокупностям правил и принципов, которые образуют *методы программирования*. Например, существуют *функциональный* и *декларативный* методы программирования. Использование языка PL/1 предполагает применение *императивного* метода, (от *imperative* — приказ) т.е. программист должен явно задать совокупность *действий*, выполнение которых должны привести к желаемому результату. Та часть программы на языке PL/1, которая не содержит действий, является описательной и предназначена лишь для передачи *компилятору* различной информации о явно указанных действиях с целью реализации их правильным и наиболее эффективным способом. Заметим, однако, что на этапе реализации все методы программирования в конечном итоге будут сведены к императивному.

Поскольку одинаковые действия можно представлять в разных формах, форма представления действий определяет *стиль* программирования. Например, фрагмент программы может быть представлен в виде вложенных друг в друга условных операторов, каждый следующий из которых вложен в альтернативную ветвь предыдущего, а можно тот же фрагмент представить, как множество операторов с индексированными метками и единственным оператором перехода по текущему индексу.

Какой бы стиль не избрал программист, существует ряд правил пригодных для всех стилей и методов:

— идентификаторам следует давать понятные имена, желательно на русском; хорошо зарекомендовал себя прием, когда на начальном этапе наброска текста программы используются короткие не содержательные имена, а затем, когда построение программы во многом уже определилось, редактором эти имена автоматически заменяются на русские и содержательные;

— должны быть содержательные неформальные комментарии, например, комментарий: счетчик +=1; // увеличиваем счетчик практически ничего не привносит в понимание текста программы, а такой комментарий — привносит: счетчик +=1; // учитываем очередное событие таким образом, большинство комментариев должны объяснять не что делает программа (это и так видно из ее текста), а зачем она это делает;

— текст программы должен быть как можно более структурирован, лучше отделить описание от операторов действия, рядом стоящие подобные операторы желательно разместить на двух строках точно друг над другом,

чтобы даже беглого взгляда было достаточно, чтобы понять их одинаковость, кстати, так легче ищутся описки;

— желательно использовать псевдографические символы для структурирования (напоминаем, они воспринимаются как просто пробелы), например, выделять циклы:

```
┌цикл i =1 с_шагом 2 до 1000;
│ x=a(i)+b(i);
│  ┌цикл j=1 до 1000;
│  │y=x*c(i, j);
│  │…
│  └конец j;
└конец i;
```

Здесь с помощью трех псевдографических символов «┌» «│» и «└» явно выделены операторы, относящиеся к циклу, причем циклы могут занимать много строк и, таким образом, на текущем экране может быть не видно ни заголовка цикла, ни его окончания; в этом случае и помогают такие линии.

Проверь себя

1. Что такое императивный метод программирования?
2. Что такое стиль программирования
3. В какие места текста программы нельзя вставить псевдографические символы?

11.3. Выполняемый модуль. Использование стандартных процедур

Краеугольным камнем работы в среде Windows является обращение к ее многочисленным стандартным процедурам (подпрограммам), которые называются API (аббревиатура от *application programming interface*). Для подключения API к программам используется базовый механизм динамически подключаемых библиотек или DLL-библиотек (аббревиатура от *dynamic link library*). Этот механизм принципиально отличается от использования библиотек процедур в терминах PL/1, когда часть или вся библиотека объединяется с кодами программы в единый выполняемый модуль. Выполняемый модуль, использующий DLL-библиотеки, содержит специальную таблицу *импорта* процедур. Операционная система загружает выполняемый модуль в память целиком и DLL-библиотеку (или несколько библиотек) в память целиком, а затем, используя данные таблицы импорта, заполняет специальные переменные в загруженном модуле адресами процедур из DLL-библиотек, причем и сами загруженные библиотеки проходят подобную настройку. С точки зрения терминов PL/1 вызов API-

процедуры происходит не для явно указанной метки входа, а вызовом переменной типа входа в процедуру, значением которой и является метка входа в API-процедуру. Это позволяет настраивать только одну переменную в выполняемом модуле для каждой API-подпрограммы, а не все подряд операторы ее вызова в модуле.

Ниже будет приведен пример, как программист может создать свою DDL-библиотеку.

Имеется два подхода к реализации выполняемого модуля.

При одном подходе для выполняемого модуля требуется специальная отдельная *программа-интерпретатор*, которая обрабатывает и выполняет коды модуля, причем эти коды могут не соответствовать командам аппаратных средств (например, *байт-код*). По существу, интерпретатор является мини-операционной системой для выполняемого модуля. Связь с основной операционной системой осуществляется путем вызовов API через механизм DLL внутри самого интерпретатора.

При другом подходе всегда имеется *системная библиотека* языка, которая является его неотъемлемой частью и уже содержит внутри себя вызовы некоторых API для реализации базовых конструкций языка. Выполняемый модуль всегда объединяется в единое целое с этой библиотекой или ее частью. Никакой отдельной программы-интерпретатора не требуется, выполняемый модуль всегда содержит реальные команды аппаратных средств. Связь с операционной системой осуществляется через вызовы процедур системной библиотеки, которые, в свою очередь, вызывают необходимые API. Но API через механизм DLL могут быть вызваны и прямо из выполняемого модуля, минуя системную библиотеку, как внешние процедуры в терминах PL/1.

В рассматриваемой версии языка реализован второй подход. При этом системная библиотека расположена внутри файла PLINK64.EXE, ее часть всегда подключается к создаваемому выполняемому модулю, составляя с ним единое целое и обеспечивая вызов всех базовых API (приведены ниже в таблице 11.1), необходимых для базовых конструкций языка. Заметим, что этих API достаточно для реализации всех примеров, приведенных выше.

ТАБЛИЦА 11.1

Список API, реализующих базовые конструкции языка PL/1

№ п/п	Имя API	Как используется в PL/1
1	CreateFileA	Открытие (создание для нового) файла
2	CreateFileMappingA	Открытие файла, отображенного на память
3	ReadFile	Чтение из файла
4	WriteFile	Запись в файл
5	CloseHandle	Закрытие файла
6	SetFilePointer	Установка текущего положения для чтения/записи файлов прямого доступа
7	GetFileSize	Установка длины файла при открытии
8	MapViewOfFile	Получение адреса файла, отображенного на память
9	GetLastError	Получение кода ошибки ОС при работе с файлами и при обработке состояний
10	GetShortPathNameA	Формирование имени в стиле MS DOS для стандартного сообщения об ошибке
11	ExitProcess	Оператор STOP/СТОП, окончание главной
12		процедуры
13	GetStdHandle	Открытие файлов стандартного ввода и вывода
14	VirtualAlloc	Выделение начальной памяти для оператора ALLOCATE/ДАЙ_ПАМЯТЬ
15	VirtualFree	Для определения объема доступной памяти
16	GetCommandLineA	Получения параметра для главной процедуры
17	SetUnhandledExceptionFilter	Для реализации операторов ON/КОГДА
18	UnmapViewOfFile	Для закрытия файла, отображенного на память
19	GlobalMemoryStatus	Для определения размера физической памяти
20	GetCurrentDirectoryA	Для реализации встроенной переменной ?ТЕКУЩИЙ_ПУТЬ (текущая папка при запуске программы)
21	AllocConsole	Для создания окна стандартного вывода
22	CreateThread	Для параллельной работы процедур
23	TerminateThread	Для снятия параллельно работающей процедуры
24	Sleep	Для реализации оператора DELAY/ЗАСНУТЬ
25	RtlAddFunctionTable	Для перехвата ошибок операционной системы и превращения их в состояния в терминах PL/1
26	GetLocalTime	Для реализации встроенных функций DATE/ДАТА и TIME/ВРЕМЯ
27	FormatMessageA	Для формирования стандартного сообщения об ошибке
	CharToOemA	Для формирования стандартного сообщения об ошибке

API CharToOemA находится в библиотеке User32.DLL, остальные в Kernel32.DLL

Таким образом, и выполняемый модуль, созданный программистом, и DLL-библиотеки, являются отдельными программами, представленными единым форматом, загруженные в память и организующие связь между собой через таблицы импорта. Кроме этой таблицы каждая программа может иметь и *таблицу экспорта*, показывающая, какими ее API могут пользоваться другие программы. В терминах языка PL/1 API — это внешние процедуры или функции, которые не надо транслировать и у которых параметры организованы принятым в среде Windows способом.

Можно также определить стиль программирования на языке PL/1 в среде Windows, как стиль, основанный на использовании (явно или неявно) API Windows.

Проверь себя

1. Что такое API-функция?
2. Что такое механизм DLL?
3. Нужен ли в рассматриваемой версии языка интерпретатор для выполняемых модулей?
4. Как происходит неявный вызов API-функций при реализации базовых конструкций языка PL/1?
5. Через какие элементы выполняемых модулей осуществляется связь между программами и DLL-библиотеками?

11.4. Вызов в языке PL/1 API как внешних процедур

Интерфейс API был разработан без учета требований языка PL/1, поэтому вызвать функцию API просто как обычную внешнюю процедуру PL/1 нельзя. Во-первых, в имени API большие и малые латинские буквы различаются (напомним, в PL/1 — нет). Во-вторых, длина имени может быть длиннее 31 символа, хотя для большинства API — не больше. В-третьих, типы параметров ограничены лишь указателями, целыми числами и короткими битовыми строками. Поэтому в рассматриваемой версии языка функции API, как внешние процедуры PL/1 должны быть описаны не только с использованием ключевых слов ENTRY/ДЛЯ_ВЫЗОВА и/или RETURNS/ВОЗВРАЩАЕТ, но и еще специально добавленного в язык ключевого слова **IMPORT/ЧУЖАЯ**. Наличие этого ключевого слова автоматически добавляет описатели EXTERNAL/ОБЩЕЕ и VARIABLE/ КОСВЕННОЕ, а также сообщает компилятору, что параметры и возможный результат работы функции должны соответствовать правилам, принятым в среде Windows. Если у процедуры имеется описатель IMPORT/ЧУЖАЯ, то в качестве входных параметров могут

использоваться и переменные или выражения символьно-строчного вида в терминах PL/1. Компилятор сам преобразует такие параметры в разрешенный для API тип указатель.

После трансляции программы, имеющей такие описания, команды операторов вызова будут учитывать правила API, однако имя внешней процедуры, помещенное в объектный модуль, будет по-прежнему соответствовать лишь правилам имен внешних процедур PL/1 и, разумеется, не может быть найдено в таком виде операционной системой в таблицах экспорта DLL-библиотек.

Поэтому в рассматриваемой версии языка редактор связей имеет таблицу соответствий имен внешних процедур по правилам PL/1 и имен API, причем там же указаны и имена DLL-библиотек, куда входят данные API-функции. Такая таблица составлена заранее, содержит свыше 9100 имен системных вызовов для наиболее часто используемых DLL-библиотек, а именно: KERNEL32, USER32, GDI32, SHELL32, WINMM, IMAGEHLP, OLE32, NTDLL, WININET, ADVAPI32, COMCTL32, GDIPLUS, WINHTTP, WSOCK32, MAPI32.

Можно считать, что имена внешних процедур по правилам PL/1 являются в этой таблице *псевдонимами* реальных имен API и, вообще говоря, могли бы выглядеть по-другому, например, по-русски. Но поскольку в документацию API Windows включены только реальные имена API, неудобно пользоваться измененными именами, поэтому псевдоним практически всегда является тем же именем, только переведенным в прописные буквы.

Используя такую таблицу, редактор связей составляет таблицу импорта, где указывается реальное имя API, DLL-библиотека, которую нужно загрузить операционной системе, а также адрес переменной типа входа в процедуру. Имя именно этой переменной составлено по правилам языка PL/1 и является псевдонимом имени API. Именно в эту переменную операционная система подставит адрес API, и именно указание этой переменной в операторе вызова приведет к передаче управления из программы на PL/1 в нужную DLL-библиотеку и в нужный вход API.

Во время своей работы редактор связей создает текстовый файл данных с расширением .SYM, где перечислены имена объектов PL/1-программы и назначенные им адреса. В конце этого файла после заголовка «?PE» перечислен список имен API-функций (их псевдонимов по правилам PL/1), которые редактор связей нашел в таблице и поместил в таблицу импорта. Если в PL/1-программе указан явный вызов API-функции, но имени соответствующего псевдонима нет в этом списке, соответствующего имени API не будет в таблице импорта, переменная не будет настроена и попытка

вызова API приведет к состоянию ошибки. Чаще всего такое случается, когда имя псевдонима описано с ошибкой.

Существует альтернативный способ вызова API-функций, когда настройка адреса переменной типа вход в процедуру происходит не в момент загрузки программы, а прямо при ее выполнении. При таком способе возможна задержка в выполнении программы, связанная с необходимостью загрузки новой DLL-библиотеки (если она еще не была загружена). Но зато этот способ универсальнее и не требует таблицы соответствий имен и псевдонимов. Для этого необходимо обратиться к служебной процедуре **?ЗАГРУЗИТЬ_ИЗ_DLL**, которая предварительно должна быть описана как

```
ОПС ?ЗАГРУЗИТЬ_ИЗ_DLL(ТЕКСТ(*) РД, ТЕКСТ(*) РД, УКАЗ) ВОЗВРАЩАЕТ(ТОЧНОЕ);
```

Пример вызова

```
?загрузить_из_dll( 'D3D9.DLL', 'Direct3DCreate9', адрес(direct3dcreate9));
```

Первый параметр этой процедуры — строчно-символьное выражение, задающее имя DLL-библиотеки. Второй параметр этой процедуры — строчно-символьное выражение, задающее имя API-функции с учетом больших и малых букв. Третий параметр — адрес переменной типа входа в процедуру, имеющий описатель IMPORT/ЧУЖАЯ. После окончания работы процедуры ?ЗАГРУЗИТЬ_ИЗ_DLL в эту переменную будет подставлен адрес входа в указанный API указанной DLL-библиотеки. Если не найдена библиотека — процедура возвращает 1, если не найдена API-функция – процедура возвращает 2. Если API-функция загружена — процедура возвращает ноль.

Приведем пример явного использования API функций. Пусть требуется текущий процент выполнения вычислений PL/1-программы, записанный в переменную *x* типа вещ, выводить в реальном времени в верхнюю строку-заголовок консольного стандартного окна вывода Windows. Это можно сделать следующими операторами:

```
опс x вещ, s текст(*) рд;
опс SetConsoleTitleA для_вызова(текст(*) рд) чужая;
писать в_строку (s) в_форме ('Выполнено', x, '%')(т, ч(6,2));
SetConsoleTitleA(s);
```

Или с английскими служебными словами

```
dcl x float, s char(*) var;
dcl SetConsoleTitleA entry(char(*) var) import;
put string(s) edit ('Выполнено', x, '%')(a, f(6,2));
SetConsoleTitleA(c);
```

Напоминаем, что малые буквы в PL/1 переводятся в большие, поэтому могли бы быть все написаны большими или малыми. Смысл оператора формирования строки S приведен в § 13.4.

Проверь себя

1. Можно ли вызвать API-функцию как обычную внешнюю процедуру PL/1?
2. Почему потребовалось вводить в язык описатель IMPORT/ЧУЖАЯ?
3. Зачем потребовались псевдонимы для имен API-функций?
4. Как вызвать API-функцию, если ее имя и DLL-библиотека неизвестны до начала работы выполняемого модуля?
5. Как по содержимому файла .SYM проверить, что имя (псевдоним) API-функции указано правильно?

11.5. Компиляция программ. Параметры компиляции

Следующим шагом после собственно написания текстов программ являет их компиляция, т.е. пропуск через *компилятор* для получения из исходных текстов *объектных модулей*. Объектный модуль содержит коды выполняемой программы, но часть адресов остается «недостроенными» и будет достроена при работе редактора связей по объединению всех объектных модулей в единый выполняемый модуль.

Поскольку все утилиты, включая компилятор с языка PL/1, находятся в единственном файле PLINK64.EXE, то при запуске этого файла из командной строки в среде Windows нужная утилита запускается по явно указанному расширению файла. Для того чтобы запустить компилятор требуется явно указать расширение файла с исходным текстом программы на языке PL/1 как .PL1 или .PLI, например PLINK64 TEST.PL1.

Если файл с исходным текстом имеет расширение, отличное от .PL1 или .PLI (например, .TXT), то тогда перед именем файла требуется указать звездочку и пробел, например, PLINK64.EXE * TEST.TXT

В процессе компиляции все сообщения выдаются на экран, но могут быть перенаправлены в текстовый файл-протокол (см. ниже). Сообщения делятся на ошибки компиляции (на экране красным цветом) и предупреждения (на экране фиолетовые). Ошибки компиляции отменяют создание объектного модуля. При предупреждениях объектный модуль создается и далее программа формально может быть запущена.

Следует отметить, что в конце работы компилятор возвращает управление операционной системе с кодом ответа 0 (объектный модуль создан) или с числом обнаруженных ошибок. Поэтому *скрипт* Windows, запускающий компилятор, может содержать проверку безошибочной компиляции, например:

```
PLINK64 TEST.PL1
IF ERRORLEVEL 1 GOTO :EOF
…
```

Имя файла с исходным текстом может содержать и путь к папке, где находится этот файл, однако получившийся объектный модуль будет всегда записан в текущую папку;

После имени файла с исходным текстом через пробел может быть указан необязательный список параметров компиляции, представляющий собой перечисление латинских букв без пробелов и/или цифры от 0 до 9, например:

```
plink64 test.pl1 qrtsd2
```

Цифра задает уровень отладочной печати, буквы определяют различные параметры или режимы компиляции, причем при выполнении программы, режимы, установленные при компиляции ее главного модуля доступны через специальную *встроенную переменную*, **?КЛЮЧИ** которую нужно описать как

```
опс ?ключи бит(32) общее;
```

Параметры в виде латинских букв могут быть заданы как в командной строке, так и установлены с помощью *системного реестра* Windows. В последнем случае, они сохраняются и действуют при последующих вызовах до новой установки в реестре. Однако если задать параметр в реестре, а затем указать его же при вызове, параметр при работе будет отменен (инвертирован).

Для установки параметров компиляции через реестр требуется в разделе «HKEY_CURRENT_CONFIG\SOFTWARE» создать имя раздела «PL/1», а в нем подразделы с названиями «A» - «Z» типа REG_DWORD и присваивать им значения 0 или 1. Дополнительно к подразделам-«буквам» подраздел с именем «%» и типом REG_SZ может содержать значение в виде одной буквы, которая заменит знак «?» в имени файла в операторе подготовки текста %INCLUDE или %ВСТАВИТЬ_ИЗ (см. § 13.6).

Некоторые параметры компиляции можно указать и в тексте самой программы буквой, перед которой стоят символы: "**%_**", например: %_R. Но так можно задавать не все параметры, поскольку некоторые действия производятся в начале работы компилятора, а не тогда, когда он встретил параметр в тексте программы.

Некоторые параметры можно отменить в теле программы командой **%_-** Например, %_-K отменяет выдачу операторов **%INCLUDE/**

%ВСТАВИТЬ_ИЗ на экран. Параметры **N, U** и **K** можно задавать и отменять несколько раз. Например, конструкция:

```
%_u /* пример текста */
%_-u
```

выведет строку «/*пример текста*/» на экран во время компиляции.

Перечислим все компиляции, допустимые в рассматриваемой версии языка.

A: ПУСК ПРОГРАММЫ

После успешной компиляции автоматически запускается редактор связи, а затем (если не было ошибок) сразу запускается полученный выполняемый модуль в отдельном окне Windows. Обычно используется для небольших программ, состоящих только из одного (главного) объектного модуля.

B: ПРЕДУПРЕЖДЕНИЕ О МЕДЛЕННЫХ ПРЕОБРАЗОВАНИЯХ

Преобразования из вещ(24) в вещ(53) проводятся в данной версии языка через промежуточное преобразование в текстовую строку, так, чтобы в новом формате мантисса была дополнена нулями в десятичном, а не в двоичном виде. Такое преобразование довольно медленно и может быть результатом ошибок описаний, а не замыслом программиста. Установленный ключ заставит компилятор выдавать предупреждение «МЕДЛЕННО» при каждом таком найденном преобразовании в программе. Результат компиляции не отменяется. Смысл данного ключа – помочь найти все такие места для возможного устранения или выноса из циклов для ускорения расчетов.

C: ВЫВОД ПСЕВДОАССЕМБЛЕРА

Вывод генерируемого кода в виде команд псевдоассемблера. С помощью специального конвертора этот код может быть преобразован в настоящий ассемблер и затем оттранслирован соответствующим транслятором, также входящим в PLINK64.EXE. Это позволяет вносить различные оптимизации «вручную» в критических фрагментах расчетов.

D: ВЫВОД ПРОТОКОЛА В .PRN

Весь поток вывода компиляции будет помещен в текущую папку в текстовый файл с расширением .PRN и таким же именем, как и имя файла с исходным текстом программы.

E: ССЫЛКА ПО ИМЕНИ ВНУТРИ РЕКУРСИИ

Если этот ключ не задан и в программе есть рекурсивные процедуры (с атрибутом RECURSIVE/РЕКУРСИВНАЯ), внутри которых, в свою очередь, есть вызов процедур, то все параметры таких вызовов передаются только по значению, т.е. для параметров создаются временные переменные. Это дает такой же эффект, как если бы Вы все параметры при вызове процедуры написали в круглых скобках, например: вызов f (x1, x2, x3); выполняется как вызов f((x1), (x2), (x3)); Ключ E отменяет это правило и внутри рекурсивных процедур можно передавать параметры по имени. Если в программе нет рекурсивных процедур, значение ключа безразлично.

F: ВЫЗОВ LINK ПОСЛЕ ТРАНСЛЯЦИИ

Если компиляция прошла успешно, то сразу после нее будет вызван редактор связей для создания выполняемого модуля. Но так можно создать модуль, если программа и состоит только из единственного главного модуля. Если компилируемая программа состоит из нескольких модулей, редактор связи просто не найдет требуемых внешних процедур, находящихся в других файлах с не заданными именами. И даже в случае единственного главного модуля также могут быть не найдены какие-либо внешние процедуры (не API). Если ключ F не задан — это означает, что пока программист и не вызывал редактор связей, поэтому сообщения о ненайденных процедурах на экран не выдаются. Если ключ F установлен — сообщения о не найденных процедурах выдаются на экран. Заметим, что вызов редактора связей для нескольких модулей — это отдельный этап обработки, прямо не зависящий от ключа F.

G: ОТМЕНА СООБЩЕНИЯ «КОНЕЦ ПРОГРАММЫ»

В конце работы каждой PL/1-программы автоматически выдается сообщение «Конец программы». Если при компиляции задать этот ключ, выдача сообщения отменяется.

H: [] ВНУТРИ КОММЕНТАРИЯ РАЗРЕШАЮТ ВЛОЖЕННЫЕ КОММЕНТАРИИ

Если установлен данный ключ и внутри комментария встретился символ "[", то до встречи символа "]" все символы "*" заменяются пробелом. Это позволяет закомментировать фрагмент программы, в котором есть свои внутренние комментарии, например:

```
/*[
X=1; /* пример внутреннего комментария */
]*/
```

I: ВЫВОД ПСЕВДОАССЕМБЛЕРА И ВЫВОД ИСХОДНОГО ТЕКСТА

Кроме строк исходного текста в протоколе будет печататься сгенерированный машинный код и его представление на языке псевдоассемблера. Используется для поиска ошибок в компиляторе.

J: ВЫВОД НЕДОСТУПНЫХ МЕСТ

Если компилятор обнаружит операторы, на которые невозможно передать управление (например, после оператора goto/идти следующий оператор не имеет метки), будет выдана информация об этом. Если ключ не установлен — таких сообщений не выдается.

K: БЕЗ ВЫВОДА %INCLUDE

При компиляции на экран не выводится служебная информация в виде операторов %INCLUDE/%ВСТАВИТЬ_ИЗ и описания ключей.

L: ВЫВОД ИСХОДНОГО ТЕКСТА

Выдача листинга с номерами исходных строк и адресацией памяти, начиная с которого расположен код, сгенерированный из данной строки. Этот режим автоматически устанавливается, если был задан ключ I. Глубина вложенности каждого блока индицируется латинской буквой.

M: РАЗРЕШЕНИЕ ПРЕДОПРЕДЕЛЕННЫХ %REPLACE/ %ЗАМЕНИТЬ

Ключ разрешает загрузить перед компиляцией программы русские эквиваленты ключевых слов. Если ключ не установлен — это позволяет отменить использование русских ключевых слов, например, когда в старых программах они уже использованы как переменные. Кроме этого, диагностические сообщения и описания переменных в протоколе выдаются на английском.

N: ПОКАЗ ВЛОЖЕННОСТИ УРОВНЕЙ

Печать уровней вложенности блоков и процедур. Каждая пара
PROCEDURE/ПРОЦЕДУРА — END/КОНЕЦ,
BEGIN/БЛОК — END/КОНЕЦ и
DO/ЦИКЛ — END/КОНЕЦ
считается за один уровень и отмечается буквами от «*a*» до «*z*». Начальный уровень главной программы — «*a*». Каждый BEGIN/БЛОК и DO/ЦИКЛ увеличивают уровень на одну букву, каждая PROCEDURE/ПРОЦЕДУРА —

на две буквы. Используется при поиске нарушенных границ блоков, циклов и процедур.

O: ОТКАЗ ОТ .OBJ

Отказ от создания объектного модуля, используется при поиске и исправлении ошибок в программах, когда конечный результат пока не нужен.

P: ВКЛЮЧЕНИЕ ВСЕХ ИМЕН

В объектный модуль включена специальная таблица (в виде записи типа LOCSYM), содержащая имена всех меток и переменных в программе. Используя эту запись, редактор связей создает файл имен .SYM в наиболее полном виде и этот файл при отладке позволит встроенному в PL/1-программу символьному отладчику установить связь между именами объектов программы и их адресами при выполнении программы.

Q: КОНТРОЛЬ ЦЕЛОЧИСЛЕННОГО ПЕРЕПОЛНЕНИЯ

Если этот ключ установлен, то после каждой арифметической операции с объектами типа FIXED BINARY/ТОЧНОЕ ДВОИЧНОЕ компилятор начнет вставлять проверки, контролирующие переполнение. В случае переполнения возникнет состояние FIXEDOVERFLOW/ЧИСЛО_НЕ_ПРЕДСТАВИМО.

R: ОТЛАДКА

В выполняемый модуль будут вставлены специальные коды, отмечающие текущие строки и имена вызываемых процедур для работы встроенной функции ONLOC. В случае аварийного окончания программы, на экран будет выдана более подробная информация (см. ниже).

S: ВЫВОД РАСПРЕДЕЛЕНИЯ ПАМЯТИ

Вывод таблицы всех переменных программы с указанием всех параметров (в том числе назначенных по умолчанию).

T: КОНТРОЛЬ ИНДЕКСОВ И SUBSTR

Этот ключ заставляет компилятор генерировать специальные команды проверки при каждом обращении в программе к элементу массива или к функции SUBSTR/ПОДСТРОКА. При выполнении программы эти команды проверяют, соответствует ли текущее значение индекса описанию массива и допустимы ли операнды в функции SUBSTR/ПОДСТРОКА. Если индекс выходит за границы массива, возникает состояние SUBSCRIPTRANGE/ВНЕ_ИНДЕКСА. Если неверен параметр подстроки, возникает состояние

STRINGRANGE/ ВНЕ_ПОДСТРОКИ. Правильность параметров подстроки определяется по следующим правилам: если длина переменной x равна l , то для подстрока(x, k, n) должны выполняться условия

$$1 <= k <= max\ (1,l)\ \text{и}$$

$$0 <= n <= l-k+1,$$

Следует отметить, что включение этих проверок может существенно увеличить код программы и замедлить ее работу.

U: ВЫВОД ПРИ КОМПИЛЯЦИИ НА ЭКРАН

Этот ключ эквивалентен по действию ключу N, однако выдает только текст исходной строки (без однострочных комментариев). Его можно использовать для выдачи произвольного текста при компиляции на экран, например:

```
%_U
/* МОДЕРНИЗИРОВАННАЯ ВЕРСИЯ */
%_-U
```

Кроме этого, данный ключ можно использовать, для того, чтобы «закомментировать» предупреждение. Каждое предупреждение при компиляции должно быть проанализировано на предмет ошибки в данном фрагменте программы. Если ошибки нет, желательно выделить фрагмент, вызывающий предупреждение этим ключом, например:

```
%_U
declare SYSPRINT file;
%_-U
```

Теперь, выделенный фрагмент текста программы будет выдаваться на экран, но предупреждающее сообщение исчезнет. Это позволит при компиляции останавливаться только на новых, еще не проанализированных предупреждающих сообщениях.

V: ВЫВОД ВНУТРЕННЕГО КОДА

Вывод результата разбора программы на промежуточном языке для контроля компилятора.

W: КОРОТКОЕ ВЫПОЛНЕНИЕ ЦЕПОЧЕК «И» И «ИЛИ»

Если в программе есть операторы типа:

```
если x1 & x2 & x3 ... тогда ...    или
если x1 | x2 | x3 ... тогда ...
```

то при отсутствии этого ключа обязательно вычисляются все выражения x_1, x_2, …, x_n. Если ключ W установлен, то генерируются дополнительные команды проверки и перехода. В первом же случае x_n = '0'Б (для операции «И») или x_n='1'Б (для операции «ИЛИ»), остальные выражения x_{n+1} уже не вычисляются, так как окончательное значение выражения сразу известно.

X: ВЫВОД НЕИСПОЛЬЗУЕМОГО

Выдача списка переменных и подпрограмм, которые описаны, но к которым нет явного обращения в программе или в подпрограммах. Звездочкой отмечены неиспользуемые процедуры, которые компилятор удалил из объектного модуля.

Y: СЛУЖЕБНЫЙ

Используется для контроля работы самого компилятора.

Z: ВЫРАВНИВАНИЕ ДАННЫХ

Выравнивание данных для повышения скорости работы программы. Объекты с нечетной длиной не выравниваются, объекты с длиной кратной слову — выравниваются по словам, объекты с длиной кратной двойному слову — выравниваются по двойным словам, объекты с длиной кратной 8 байтам — выравниваются по 8 байт.

По умолчанию заданы ключи H, J, M, P, W, Z.

Кроме 26 перечисленных параметров компиляции, возможно задание в командной строке еще уровня отладочной печати (0 — 9). Данное средство, похожее на условную трансляцию, не обязательно использовать только для отладочной печати. Однако часто требуется вывод какой-либо отладочной информации, который желательно включать и выключать, не меняя текст самой программы. Для этого можно заранее в тексте программы (обязательно в начале строк) расставить конструкции из символа процента и цифры от 0 до 9, например:

```
…
%1 писать с_новой в_виде('отладка 1');
…
%2 писать с_новой в_виде('отладка 2');
```

По умолчанию эти строки будут рассматриваться как однострочные комментарии. Если при трансляции задать ключ 1 или 2, например:

PLINK64 TEST.PL1 1 или PLINK64 TEST.PL1 2 то в исходном тексте появится или первый оператор примера или сразу оба.

Идея здесь в том, чтобы задать уровень, до которого нужно транслировать строки. Сами строки, естественно, могут содержать любые операторы, а не только печать. Процент и ноль просто заменяются на пробелы (т.е. такая строка транслируется всегда).

Необходимый уровень можно задать и в самой программе конструкцией из процента, вопросительного знака и цифры от 0 до 9. Например, если задать в начале программы %?9, все строки, начинающиеся с %0-%9 будут транслироваться.

Проверь себя

1. Чем принципиально отличается объектный модуль от выполняемого?
2. Как различаются сообщения об ошибках и предупреждения во время компиляции? Можно ли запускать выполняемый модуль, если при компиляции выдавались предупреждения?
3. Как определить, прошла ли компиляция удачно, если запуск компиляции осуществляется через «скрипт» Windows?
4. Есть ли незадействованные латинские буквы среди параметров компиляции?
5. Как исключить выдачу конкретных предупреждающих сообщений, чтобы они не отвлекали внимание от новых сообщений?
6. Что такое уровни отладочной выдачи и как их задать?
7. Могут ли некоторые параметры компиляции задаваться прямо внутри текста PL/1-программы?
8. По умолчанию выполняются ли все цепочки логических И и ИЛИ до конца в случае, когда при очередном вычислении конечный результат уже известен? Каким параметром изменяется это правило?
9. В какую встроенную переменную помещаются все параметры, заданные при компиляции данного главного модуля?

11.6. Этап редактирования связей. Редактор связей

После того, как все файлы с исходными текстами пропущены через *компилятор* и получены объектные модули, необходимо с помощью специальной программы — *редактора связей* настроить адреса внешних процедур, т.е. связей между модулями. Попутно редактор связи может также «достроить» ряд адресов, которые неудобно сразу формировать компилятору,

например, некоторые переходы на метки вперед. Редактор связей читает объектные модули в заданном порядке, их код загружает в память друг за другом, моделируя вид будущего выполняемого модуля, формирует незаполненные адреса. Получившийся в памяти образ выполняемого модуля записывается в единый файл и оформляется по правилам программ Windows, в частности формируются заголовок и таблицы импорта API. Директива запуска редактора связей имеет вид

PLINK64 $m_0 = m_1, m_2, \ldots, m_n\ p$

где m_i ($i=1, 2, \ldots, n$); $n>0$; i-ый объектный модуль или библиотека объектных модулей, p — дополнительные необязательные параметры, имеющие вид или ***/DEB*** или ***/GRAPH*** или ***/DLL***.

Необязательная конструкция $m_0=$ задает в явном виде имя выполняемого модуля (в этом случае обязательно указывать расширение .EXE), иначе в качестве имени выполняемого модуля будет взято имя первого указанного объектного модуля. Главный модуль не обязательно указывать первым. Например, пусть имеется два объектных модуля T1.OBJ и T2.OBJ, причем T1 — главный модуль программы. Тогда директивой

PLINK64 T1.OBJ, T2.OBJ

будет создан выполняемый модуль T1.EXE

Директивой PLINK64 T2.OBJ, T1.OBJ

будет создан выполняемый модуль T2.EXE

Директивой PLINK64 T3.EXE=T1.OBJ, T2.OBJ

будет создан выполняемый модуль T3.EXE, причем все три выполняемых модуля будут эквивалентны.

Группы объектных модулей с помощью специальной программы — *редактора библиотеки* могут быть объединены в файлы-библиотеки объектных модулей. Такие библиотеки могут быть также указаны в списках объектных модулей, как и отдельные объектные модули. Расширение у библиотек объектных модулей всегда .L86, например:

PLINK64 T3.EXE=T1, T2, T3.L86

Обратите внимание, что расширение .OBJ можно опускать, оно подразумевается по умолчанию. Но все другие расширения (.PL1, .L86, .EXE) должны всегда указываться явно.

Существует еще одна форма запуска редактора связей, при которой вся командная строка (кроме самого имени PLINK64) сначала записывается в текстовом файле с расширением .INP, а собственно при вызове указывается только имя этого файла, например, если в текстовый файл L.INP записать строку

```
T3.EXE=T1, T2, T3.L86
```

То директивы

```
PLINK64 T3.EXE=T1, T2, T3.L86
PLINK64 L[I]
```

Станут эквивалентны. Параметр I квадратных скобках, указывает, что задано не имя объектного модуля, а имя файла со списком модулей. Такая форма удобна при длинном списке модулей, который как угодно может быть разбит по строкам файла .INP. Если строка начинается с точки с запятой, она пропускается (превращается в комментарий). Таким образом, легко менять список модулей при запуске редактора связей всегда одной и той же директивой.

Проверь себя

1. Что такое редактор связи, и каково его основное назначение?
2. Можно ли в списке объектных модулей указывать главный модуль не первым?
3. Обязательно ли указывать расширение .OBJ для списка объектных модулей?
4. Обязательно ли указывать расширение .L86 для файлов-библиотек объектных модулей?
5. Как будет называться выполняемый модуль при редактировании связей директивой PLINK64 M1, M2.L86, M10, если файл M10.OBJ — главный модуль программы?
6. Если список объектных модулей задается не в командной строке, а в файле с расширением .INP, можно ли исключать из редактирования связей некоторые модули, не убирая их имен из текстового файла .INP?

11.7. Объединение объектных модулей в библиотеки

В ряде случаев удобно хранить один раз оттранслированные объектные модули не «россыпью» в отдельных файлах, а в одном или нескольких файлах — *библиотеке объектных модулей*. С библиотекой можно проводить различные действия — добавлять, убирать или заменять объектные модули. Библиотеки можно объединять или, наоборот, выделять часть одной библиотеки. Можно создать файл-каталог библиотеки и файл перекрестных ссылок у заданной библиотеки. Все эти действия, а также создание библиотек выполняет специальная программа — *редактор библиотеки*.

Редактор библиотеки, как и все другие утилиты, расположен в одном файле и вызывается директивой

PLINK64.EXE # m_0= m_1, m_2, …, m_n p

где m_i (i=1, 2, …, n); n>0; i-ый объектный модуль или библиотека объектных модулей, p — дополнительные необязательные параметры в квадратных скобках.

Если при вызове расширение файлов не указано, то по умолчанию принимается до знака равенства — .L86, после знака равенства — .OBJ. Если знака равенства нет — расширение по умолчанию .OBJ. При этом редактор работает с файлами .OBJ и .L86 совершенно одинаково, и определяет модуль или библиотеку по содержимому, а не по расширению.

Кроме имен файлов, можно указывать параметры p в квадратных скобках. Названия параметров можно сокращать до любой длины, но так, чтобы не получилось двусмысленности.

После вызова, редактор читает указанные файлы и создает файл-библиотеку, файл перекрестных ссылок и файл-каталог модулей. При этом файлы, имеют следующие расширения:

.INP — файл, где записана командная строка;
.L86 — исходный или результирующий файл-библиотека;
.MAP — файл-каталог модулей;
.OBJ — файл исходного объектного модуля;
.XRF — файл перекрестных ссылок.

Для создания библиотеки в командной строке указывается имя библиотеки, затем знак равенства и список включаемых в библиотеку файлов через запятую, например: PLINK64 # NEWLIB=A, B, C

Файлы с объектными модулями могут идти в любом порядке.

Для того чтобы дописать файлы в имеющуюся библиотеку, необходимо в командной строке указать имя этой библиотеки справа и слева от знака равенства: PLINK64 # MATH=MATH.L86, SIN, COS, TAN

Можно заменить один или несколько модулей в библиотеке, не меняя самой библиотеки. Общий вид директивы в этом случае:

PLINK64 # «новое имя» = «старое имя» [REPLACE [«список»]]

где «новое имя» — имя, под которым создается библиотека, «старое имя» — имя библиотеки, в которой заменяются модули, старое и новое имя может быть одним и тем же; «список» — список пар имен через запятую в виде: «имя модуля»= «имя файла», если имя файла не совпадает с именем первой метки процедуры в файле. Например,

PLINK64 # MATH=MATH.L86 [REPLACE [SQRT=NEWSQRT]]

здесь в библиотеке MATH тело модуля SQRT будет заменено на модуль из файла NEWSQRT.OBJ. Однако если «имя модуля» и «имя файла» совпадают, имя файла можно не указывать, например:

PLINK64 # MATH=MATH.L86 [REPLACE [SQRT]]

Для того чтобы убрать модуль из библиотеки, задается директива:

PLINK64 # «новое имя»= «старое имя» [DELETE [«список»]]

где «список имен» - список имен убираемых из библиотеки модулей через запятую, если это ряд соседних модулей, можно указать начальное и конечное имена через знак «минус». Например,

PLINK64 # MATH=MATH.L86 [DELETE [ADD, SUB, MUL, DIV]]

PLINK64 # MATH=MATH.L86 [DELETE [ADD-DIV]]

Переписать модули из библиотеки в библиотеку можно следующей директивой:

PLINK64 # «новая библиотека»= «старая библиотека»[SELECT [«список»]]

как и в предыдущем случае «список» - список имен модулей через запятую, или граничная пара имен через «минус». Получившуюся новую библиотеку можно склеить с другими библиотеками директивами добавления, например:

PLINK64 # ARITH=MATH.L86 [SELECT [ADD, MULT]]

однако нельзя одновременно использовать REPLACE и SELECT вместе с DELETE.

Кроме перечисленных действий с библиотеками, можно создать файл перекрестных ссылок директивой

PLINK64 # «имя» [XREF]

при этом будет создан файл с именем «имя».XRF и содержит список всех глобальных и внешних объектов, а также имена *сегментов* (общих переменных и процедур в терминах PL/1)в алфавитном порядке. Такой файл может быть создан для одного или для нескольких модулей или библиотек. Модуль, в котором описано данное имя, отмечается знаком #, после имен сегментов ставится их тип в косых скобках, например, /CODE/. В конце списка указывается число всех модулей.

Можно также создать файл-таблицу всех объектов библиотеки с помощью директивы

PLINK64 # «имя» [MAP]

Такой файл будет содержать каталог — список модулей в алфавитном порядке, после каждого имени модуля идет список имен сегментов (общих с другими модулями элементов) модуля с указанием их длины. В каталоге также имеется список глобальных объектов (PUBLIC) и внешних объектов (EXTERNAL). Если указать параметр NOALPHA, имена будут не в алфавитном порядке, а так как они встречаются в библиотеке, например:

PLINK64 # MATH.L86 [MAP, NOALPHA]

Поскольку полные каталоги обычно не требуются, можно создать частичный каталог, в котором будут указаны только имена только модулей или сегментов или только внешних объектов. Такие частичные каталоги создаются директивами:

PLINK64 # «имя» [MODULES]
PLINK64 # «имя» [SEGMENTS]
PLINK64 # «имя» [PUBLICS]
PLINK64 # «имя» [EXTERNALS]

Как и в случае работы с редактором связей, вместо списков модулей в командной строке их можно записать в текстовый файл с расширением .INP и затем запустить редактор библиотеки директивой:

PLINK64 # «имя» [INPUT]

например, пусть в файле MATH.INP записаны строки

```
math=add, sub, mul,
;---- комментарий ----
div, cos, sin,
sqrt
```

тогда обращение к редактору для создания MATH.L8:

PLINK64 # MATH [INPUT]

При таком запуске редактора можно указывать и другие директивы:

PLINK64 # MATH [IN, MAP, XREF].

Кроме этого, можно указать параметр ECHO, выдающий на экран содержимое файла .INP.

Проверь себя

1. В каких случаях целесообразно использовать библиотеки объектных модулей?
2. Какой символ определяет запуск редактора библиотеки из файла PLINK64.EXE?
3. Для чего предназначены директивы редактора библиотеки REPLACE, DELETE, SELECT? Можно ли их имена сократить до одной первой буквы?
4. Как объединить две библиотеки с именами LIB1.L86 и LIB2.L86 в одну, сохранив название объединенной библиотеки как LIB1.L86?
5. Как получить список перекрестных ссылок объединенной библиотеки LIB1.L86? Как определить в этом списке, в каком модуле находится заданное имя?

6. Как с помощью редактора библиотеки определить, в какой из библиотек (пусть с именами LIB1, LIB2, …LIB10) находится заданная внешняя процедура?

12. ЗАПУСК И ОТЛАДКА ВЫПОЛНЯЕМЫХ МОДУЛЕЙ

12.1. Понятия отладки и тестирования. Отладочные средства

После окончания подготовки программ к исполнению, т.е. после этапов компиляции и редактирования связей в большинстве случаев требуется провести этапы *отладки* и *тестирования*, которые, в свою очередь, могут потребовать изменения текста программ и повторной компиляции и редактирования связей. Этап тестирования в данном пособии не рассматривается, так как он, в основном, определяется сутью прикладной задачи. Отладка же, как процесс поиска и исправления ошибок в коде программы, рассматривается, во-первых, потому, что этот этап больше связан с языком программирования, чем с сутью решаемой задачи, а, во-вторых, в рассматриваемой версии языка имеется специальная *программа-отладчик*, призванная ускорить процесс отладки, хотя этот же отладчик может быть использован и на этапе тестирования, и на этапе эксплуатации программы.

При отладке программист может использовать в качестве отладочных средств как имеющиеся конструкции языка (явно заданные проверки, отладочные печати, обработка возникающих состояний ошибки), так и специальные средства. В качестве специального средства отладки в рассматриваемой версии языка имеется:

— утилита проверки межмодульных связей;

— специальный интерактивный отладчик.

Также в качестве отладочных средств могут быть использованы файлы перекрестных ссылок, создаваемые редактором библиотеки и назначение специальных типов переменным, которые в программе имеют физический смысл и могут быть представлены в терминах Международной системы измерений СИ (см. § 13.3).

Проверь себя

1. Охарактеризуйте все возможные средства, которые программист может использовать на этапе отладки программ.
2. В чем преимущества интерактивного режима отлаживания программы по сравнению с режимом, когда программист не вмешивается в ход исполнения программы?

12.2. Утилита проверки межмодульных связей

Для проверки соответствия общих данных и параметров процедур, описанных в разных файлах (для одного проекта) с исходным текстом можно

использовать простую утилиту сравнения описаний. При своей работе она использует тот факт, что на этапе компиляции могут быть созданы файлы-протоколы (путем задания параметров компиляции S и D), содержащие полные описания всех объектов, используемых в программе. В этом случае возможно составить общую таблицу из всех имеющихся таблиц и проверить, что все объекты с описателем EXTERNAL/ОБЩЕЕ во всех протоколах описаны одинаково. Утилита запускается директивой PLINK64 @

После запуска утилита ищет в текущей папке и во всех вложенных папках файлы-протоколы компиляции (с расширением .PRN), откуда читает все внешние данные и параметры процедур, после чего просто сравнивает их между собой на тождество. Протокол работы утилиты выдается на экран.

Таким образом, для проекта, состоящего из большого числа PL/1-модулей, можно быстро и надежно убедиться в правильном использовании процедур и общих данных.

Обратите внимание, что компиляция с ключами S и D может замаскировать ошибки компиляции, в том смысле, что они тоже будут выведены не на экран, а в файл с расширением .PRN.

Рекомендуем в таких случаях использовать «скрипт» и выполнять компиляцию два раза: сначала с выводом ошибок на экран, а если их не было, то повторять компиляцию уже с ключами S и D, например:

```
PLINK64 TEST.PL1
IF ERRORLEVEL 1 GOTO :EOF
PLINK64 TEST.PL1 SD
PLINK64 TEST /DEB
```

На этапе редактирования связей такую проверку провести нельзя, поскольку редактор связей имеет дело только с адресами и идентификаторами (внешним представлением имен). Информация же о типе данных в терминах PL/1 отсутствует в файлах объектных модулей.

Однако если на этапе компиляции были созданы файлы-протоколы с полным описанием всех объектов каждого модуля, на этапе редактирования связей можно задать директиву **/DEB**, которая укажет редактору связей, что нужно провести дополнительно еще обработку всех имеющихся файлов-протоколов и дополнительно к файлу .SYM составить файл .TRF, содержащий информацию об объектах программы в терминах PL/1.

Например, после директивы PLINK64 T1, T2 /DEB будет создан выполняемый модуль T1.EXE, файл, содержащий идентификаторы и их адреса T1.SYM и файл, содержащий информацию о типах объектов в терминах PL/1 — T1.TRF. После запуска выполняемого модуля T1.EXE и вызова отладчика (см. ниже), отладчик не только читает данные об адресах из файла

T1.SYM, но и данные о типах из файла T1.TRF, что расширяет возможности отладки и дает возможность интерпретировать участки памяти с известными адресами правильным образом (как целые и вещественные числа, строки и т.д.).

Проверь себя

1. Почему компилятор не может проверить, что вызываемая в данном модуле внешняя процедура может быть описана не с теми параметрами, чем в описании в том модуле, где она расположена?
2. Почему редактор связей не может провести такие проверки? Какие данные нужны для этой проверки?
3. Объекты с каким описателем рассматривает утилита проверки межмодульных связей? Файлы с каким расширением она читает?
4. Для чего нужен дополнительный файл данных о типах с расширением .TRF?

12.3. Встроенный интерактивный отладчик. Вывод информации

Интерактивный встроенный символьный отладчик является неотъемлемой частью рассматриваемой версии языка и предназначен для отладки программ в режиме непосредственной обратной связи с программистом. Отладчик позволяет выполнять программу по командам, по подпрограммам или по строкам исходного текста, останавливаясь в заданных местах и при заданных условиях и показывая при этом имена переменных и меток и/или их текущее содержимое.

Отладчик представляет собой служебную подпрограмму, которую редактор связи автоматически добавляет в каждый собираемый им выполняемый модуль (EXE-файл). Отладчик может быть:

— активирован при запуске программы;
— активирован при возникновении исключения в программе;
— не активирован и не влияет на работу программы.

Для работы отладчика необходимы файлы .SYM и .TRF, указанные выше. Если такие файлы отсутствуют, отладчик может быть запущен, но не сможет показать ничего, кроме выполняемого кода без меток и имен.

Приведем пример, пока, не объясняя смысл директив отладчика. Пусть в программе имеется переменная *x*, описанная как опс x(5) вещ(53); а в программе имеется оператор цикла цикл i=1 до 5; x(i)=i; конец;

Если файл .TRF не создавался, а имеется только .SYM, то если запустить программу с отладчиком и остановиться после выполнения этого оператора

цикла, отладчик сможет показать переменную *X* только в шестнадцатеричном виде: D .X

```
002B:0040A630 00 00 00 00 00 00 F0 3F-00 00 00 00 00 00 00 40
002B:0040A640 00 00 00 00 00 00 08 40-00 00 00 00 00 00 10 40
002B:0040A650 00 00 00 00 00 00 14 40-05 00 00 00 00 00 00 00
002B:0040A660 00 00 00 00 00 00 00 00-00 00 00 00 FF FF BF FF
002B:0040A670 74 A6 00 00 19 00 00 00-E7 1A 00 00 00 00 00 00
002B:0040A680 00 0D 0A 8A AE AD A5 E6-20 AF E0 AE A3 E0 A0 AC
002B:0040A690 AC EB 24 0D 0A 91 92 85-90 92 20 91 8F 88 91 8E
002B:0040A6A0 8A 20 94 80 89 8B 8E 82-24 0D 0A 8D 85 20 82 9B
002B:0040A6B0 84 85 8B 85 8D 80 20 8F-80 8C 9F 92 9C 24 66 06
002B:0040A6C0 00 00 0A 12 40 00 00 00-00 00 00 00 00 00 00 00
002B:0040A6D0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
002B:0040A6E0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

Если же файл .TRF создан, то теперь отладчик показать переменную в виде объекта PL/1: D .X

```
002B:0040A630 .X(1:5)   FLOAT(53)
         1.00000000E+000         2.00000000E+000         3.00000000E+000
 4.00000000E+000
         5.00000000E+000
```

Поскольку отладчик является подпрограммой, «зашитой» в каждый выполняемый модуль, то для работы с ним требуется лишь его активизация (и наличие .SYM и .TRF файлов).

На практике отладчик обычно вызывают в двух случаях:

— перед выполнением программы, чтобы поставить контрольные точки, посмотреть начальные значения переменных и т.п.;

— при аварийном завершении программы, чтобы проанализировать текущие значения переменных.

Для этих двух случаев предусмотрены ключи «**??**» или «**?**», которые ставятся в начале командной строки при вызове выполняемого модуля.

Используя ключ «??», программист сразу попадает в интерактивный режим отладчика до начала выполнения операторов программы, но уже после работы служебной подпрограммы подготовки к исполнению программы, которая, например, выделяет память и собственно активизирует отладчик.

С ключом «?» программа начинает выполняться обычным образом, а в случае возникновения состояния ошибки (или особого случая) вызывается отладчик. При этом если программа завершилась успешно — отладчик так и не вызывается. Можно искусственно вызвать особый случай отладки и попасть в отладчик, если написать в PL/1 программе оператор МАШ_КОД('CC'B4);

Если в командной строке этих ключей не задано, отладчик не используется и никак не влияет на работу программы. В этом случае можно использовать «внешние» отладчики среды Windows типа WinDbg. К сожалению, работать одновременно с двумя и более отладчиками невозможно.

Например, если вызов программы производится строкой

```
PROGRAM 1 2 3 4 5 'A'
```

то вызов отладчика до выполнения программы:

```
PROGRAM ?? 1 2 3 4 5 'A'
```

а вызов отладчика при аварийном завершении:

```
PROGRAM ? 1 2 3 4 5 'A'
```

Обратите внимание, при запуске программы первые символы ? или ?? исключаются из командной строки и с точки зрения PL/1-программы доступная ей командная строка (как входной параметр главной процедуры) никогда не содержит таких символов. Сами символы ? или ?? необязательно должны отделяться пробелом от остальной командной строки.

В результате активизации отладчика в начале программы, или в случае возникновения исключения при работе программы, происходит переход в интерактивный режим отладки, о чем говорит приглашение «-» и вывод стандартной информации, например:

```
Загрузка файла ссылок
#83
AX=0000  BX=0030  CX=7764  DX=7765  SP=0012  BP=0000  SI=0041  DI=0030
   0001     0014     7FA8     1DB0     FF48     0000     3200     3320
R8=0065  R9=0000 R10=0000 R11=7765 R12=0000 R13=0000 R14=0000 R15=0000 CTP
   0430     0012     0200     1DB0     0000     0000     0000     0000
CS=0033  DS=002B  SS=002B  ES=002B  FS=0053  GS=002B  IP=00401214  --I------
00401214 C7053A94000005000000      MOV    D PTR [0040A658],00000005  DS:0040
A658=00000000
```

Здесь выдано сообщение, что успешно загрузился файл ссылок (.SYM), а также к нему добавлено 83 объекта из файла .TRF. Таким образом, у отладчика имеются и адреса переменных и их типы.

Стандартная информация состоит в выводе текущего состояния регистров процессора и кода текущей команды с ее дисассемблированием. При этом данная команда еще не выполнилась и может быть выполнена при продолжении работы. Выводится также состояние флагов процессора, а если компиляция была с ключом R, то выводится также номер текущий строки исходного текста текущего модуля и первые две буквы его имени.

Обратите внимание, что выводятся только младшие половины 8-байтных регистров. В большинстве случаев в старших 4 байтах содержатся нули. Признаком нулей в старших половинах служит знак равенства, отделяющий имя регистра от его содержимого. Если старшие 4 байта содержат не нули, знак равенства заменяется на знак неравенства. Посмотреть весь регистр можно отдельной командой. Такой прием позволяет не забивать экран ненужной информацией, поскольку даже если старшая половина регистра содержит не нули, а часть адреса, эта часть меняется редко.

В примере выше, все старшие части всех регистров содержали нули.

В следующем примере отладчик остановился на 685 строке модуля, имя которого начинается с букв SI и старшая половина регистра RAX не равно нулю.

```
AX#0000  BX=01B5  CX=7764  DX=7765  SP=0012  BP=0000  SI=0042  DI=01B5
   0001     0015     7FA8     1DB0     FF48     0000     0200     4B87
R8=001C  R9=0000 R10=0000 R11=7765 R12=0000 R13=0000 R14=0000 R15=4953 CTP SI
   0430     0012     0200     1DB0     0000     0000     0000     02AD   685
CS=0033  DS=002B  SS=002B  ES=002B  FS=0053  GS=002B  IP=00401210  --I------
00401210 BEBC4B4100                  MOV    ESI,00414BBC
```

Регистр RAX можно посмотреть (и изменить) отдельной командой:

-rax

AX FFFFFFFF'00000001 .

Стандартная информация, которая выдается при каждом входе в отладчик, также может быть запрошена в любой момент по команде R без параметров.

Проверь себя

1. В каких режимах инициализации можно использовать встроенный отладчик?
2. Как должна выглядеть командная строка запуска выполняемого модуля (пусть с именем T1.EXE) в случае, если нужно войти в отладчик до начала исполнения программы? После возникновения состояния ошибки?
3. Можно ли в тексте программы написать условный оператор так, что если условие не выполняется, управление передается в отладчик?
4. Где в стандартной информации отладчика расположен текущий номер строки исходного текста и первые две буквы названия модуля? Какой параметр при компиляции нужно задать, чтобы эта информация выдавалась?

12.4. Интерактивные команды отладчика

В интерактивном режиме можно вводить различные команды отладчика, тем самым, влияя на состояние программы и ее работу. Большинство команд совпадает с «классическим» набором команд отладки, присутствующим почти во всех отладчиках и восходящим еще к временам MS DOS.

В большинстве команд отладчика в качестве операндов задаются адреса памяти. Адреса могут задаваться шестнадцатеричными числами, однако в данном символьном отладчике операнды могут быть заданы и несколькими другими способами:

— как имена регистров (всегда с буквы E, например, EBX, реально будет взят RBX, но задать напрямую регистры R8-R15 нельзя);

— как имена переменных из PL/1-программы, тогда начинаются с точки;

— как десятичное число – тогда начинается с символа #;

— как символ ^, тогда означает содержимое по адресу из стека, т.е. [RSP], два ^^ - как [RSP]+8, ^^^ - как [RSP]+16 и т.д.;

— как **@** — тогда это признак косвенной адресации;

— как сумма или разность двух операндов;

— как символьная строка из одного или более символов в кавычках (для некоторых команд, но это не адрес).

Несколько примеров. Пусть нужно задать операнд 401000 (шестнадцатеричное), допустим, это и адрес переменной НАЧАЛО, тогда операнд может выглядеть как:

401000 (в шестнадцатеричном виде по умолчанию)
#4198400 (в десятичном виде)
.НАЧАЛО (в символическом виде)
ESI (через регистр, если в нем записано значение 401000)

Адрес может быть выражением:

400100+2
#256-#64
.НАЧАЛО-.КОНЕЦ
ESI+EDI
^
^^
@.НАЧАЛО

Почти все команды состоят из одной латинской буквы и параметров после нее. Ряд команд может иметь префикс «-». Ниже описываются команды, но не в алфавитном порядке, а в порядке их важности.

Команда **Q**. Выход из отладчика

Эта команда не имеет параметров. Выход из отладчика произойдет также, если закончилась отлаживаемая программа самостоятельно или по клавишам CTRL+C. С окончанием отладчика немедленно оканчивается и отлаживаемая программа.

Команда **?** вывода подсказки

Краткий перечень команд выдается на экран, если ввести символ «?», а если ввести два этих символа подряд, выдается краткое описание форматов команд.

Команда **D**. Просмотр памяти

Команда имеет вид:

D[W/D/F/FF] «начало» «конец» или

D[W/D] «начало» + «число»

где «начало» — это начальный адрес вывода, «конец» — конечный адрес вывода, «число» — число выводимых байт.

Содержимое памяти выводится в виде шестнадцатеричных кодов и соответствующим им текстовых строк. Если очередной байт имеет код меньше 20 или 7F или FF, в строке он заменяется на символ точки. Содержимое памяти может выдаваться байтами **(**команда D), словами (DW) двойными словами (DD), в формате чисел с плавающей точкой (DF) и в виде чисел с плавающей точкой двойной точности (DFF)

Если «конец» или «число» не указаны, по умолчанию выводится 12 строк по 16 байт. (132 байта). Эту порцию можно изменить директивой

-D «число»,

например: -D 80 означает вывод по умолчанию 128 байт (80 шестнадцатеричное).

Если «начало» не указано, вывод идет с текущего адреса данных (этот адрес запоминается до следующей директивы D). Первоначально адрес по умолчанию равен 400000, что соответствует обычному началу доступной PL/1-программе памяти.

По команде DF (DFF) содержимое памяти представляется в формате обычной или двойной точности IEEE-754. При этом каждое число занимает в памяти соответственно 4 и 8 байт. Примеры:

```
-d 400000

002B:00400000 4D 5A 90 00 03 00 00 00-04 00 00 00 FF FF 00 00 MZP.............
002B:00400010 B8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 00 ╕.......@.......
002B:00400020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ................
002B:00400030 00 00 00 00 00 00 00 00-00 00 00 00 70 00 00 00 ............p...
002B:00400040 0E 1F BA 0E 00 B4 09 CD-21 B8 01 4C CD 21 54 68 ..╢..┤.=!╕.L=!Th
002B:00400050 69 73 20 70 72 6F 67 72-61 6D 20 63 61 6E 6E 6F is program canno
002B:00400060 74 20 62 65 20 72 75 6E-20 69 6E 20 44 4F 53 20 t be run in DOS
002B:00400070 50 45 00 00 64 86 03 00-F9 D9 F7 60 00 00 00 00 PE..dЖ..·┘ў`....
002B:00400080 00 00 00 00 F0 00 22 00-0B 02 09 00 00 E1 01 00 ....Ё."......c..
002B:00400090 00 00 00 00 00 00 00 00-00 10 00 00 00 10 00 00 ................
002B:004000A0 00 00 40 00 00 00 00 00-00 10 00 00 00 02 00 00 ..@.............
002B:004000B0 06 00 01 00 06 00 01 00-06 00 01 00 00 00 00 00 ................
```

```
-dw 400000 +1f

002B:00400000 5A4D 0090 0003 0000-0004 0000 FFFF 0000         MZP.............
002B:00400010 00B8 0000 0000 0000-0040 0000 0000 0000         ╕.......@.......

-dd 400000 +1f

002B:00400000 00905A4D 00000003-00000004 0000FFFF     MZP.............
002B:00400010 000000B8 00000000-00000040 00000000     ╕.......@.......

-dd esp

002B:0012FF48 0040C0E0 00000000-00401007 00000000     pL@.......@.....
002B:0012FF58 7731556D 00000000-00000000 00000000     mU1w............
002B:0012FF68 00000000 00000000-00000000 00000000     ................
002B:0012FF78 00000000 00000000-00000000 00000000     ................

000000000040C0E0 0040C0E0 .?STOPX
```

Просмотр памяти через операнд регистр ESP (реально RSP) приводит к дополнительным действиям. Каждые 8 байт с вершины стека и вниз (всего 100H) проверяются на попадание в таблицу адресов подпрограмм, и потом отдельно выдается список адресов и ближайшие адреса меток и подпрограмм к ним. В данном примере в стеке найден адрес, строго совпадающий со служебной подпрограммой ?STOPX. Если используется другой операнд, не ESP, таких дополнительных действий не выполняется.

Это сделано для упрощения анализа адресов возврата в стеке. Заметим также, что Windows требует всегда кратности стека 8. Попытка нарушить отладчиком кратность стека 8 приведет к краху и отладчика и всей задачи.

```
-dff 400000

002B:00400000 6.37066138E-314 1.39064994E-309 9.09080788E-322 3.16202013E-322
002B:00400020 0.00000000E+000 0.00000000E+000 0.00000000E+000 2.37663528E-312
002B:00400040-1.32170658E+063 3.67404926E+194 1.25013747E+243 5.76720549E+228
002B:00400060 1.24033552E+224 5.76071297E-153 4.90206491E-309 8.03773625E-315
002B:00400080 5.00743473E-308 2.61294074E-309 0.00000000E+000 8.69169476E-311
002B:004000A0 2.07226151E-317 1.08646184E-311 1.39079848E-309 3.23820505E-319
```

В последнем примере показан вывод памяти в формате IEEE-754. Наличие огромных степеней практически всегда показывает, что реально в этой области памяти находятся вовсе не числа с плавающей точкой.

Команда **K** установки контрольной точки

В программе можно указать несколько мест, где необходимо остановить выполнение (например, чтобы посмотреть содержимое памяти). Такие места называются контрольными точками. Это должны быть адреса каких-то команд в программе, а, чаще всего, это будут адреса меток в программе. Программист может заранее расставить метки в тех местах программы, где интересно

остановиться. «Лишние» метки практически не влияют на работу выполняемого модуля.

Контрольные точки могут быть программные и аппаратные. Установка программной контрольной точки заключается в том, что в код команды по указанному адресу помещается специальная однобайтовая команда INT 3, при выполнении которой, управление попадет в отладчик. Таким образом, остановка программы и выход в отладчик происходит до выполнения команды по адресу контрольной точки. Аппаратные контрольные точки используют отладочные регистры процессора. Использование аппаратных контрольных точек позволяют не только остановиться в любом месте программы, но и остановиться по событию чтение/запись заданной ячейки памяти. Следует иметь в виду, что остановка по контрольной точке по данным происходит после выполнения команды, выполнявшей чтение или запись. Однако аппаратная остановка по команде происходит тоже до выполнения команды.

Программные контрольные точки могут быть оперативные и постоянные. Оперативных контрольных точек может быть одна или две, и они указываются при каждом запуске или продолжении отлаживаемой программы командой G (см. ниже). Такие точки автоматически снимаются, если управление снова вернулось в отладчик, независимо от того, произошла ли остановка из-за того, что программа дошла до одной из этих точек или попала на одну из постоянных точек.

Постоянные контрольные точки устанавливаются и снимаются с помощью соответствующих команд, обычно перед запуском программы. Одновременно можно установить 16 постоянных программных контрольных точек и четыре постоянных аппаратные точки.

Вид команды: K [Z] «адрес» [«число раз»] [тип] [размер] [условие]

Здесь Z необязательный признак звонка. Он позволяет установить контрольную точку со звонком. При прохождении программы через эту точку отладчик не будет выдавать на экран стандартную информацию, а вместо этого выдаст звуковой сигнал. Заметим, что можно регулировать длительность звонка, задав в команде не одну, а две и больше букв Z подряд.

«адрес» — адрес команды или переменной (обычно символьное имя), на которую устанавливается данная точка.

«число раз» — счетчик проходов через эту точку до остановки (по умолчанию равен 1).

«тип» — или K или W или R — один из трех возможных типов аппаратных контрольных точек (если тип не указан — задается программная контрольная точка):

K — аппаратная контрольная точка по команде по заданному адресу;

W — аппаратная контрольная точка по записи по заданному адресу;

R — аппаратная контрольная точка по чтению или записи. по заданному адресу.

«размер» — для контрольных точек по данным, задает размер объекта 1, 2 или 4 байта (по умолчанию 4).

«условие» — для контрольных точек по данным можно задать условие сравнения с константой. Для этого нужно задать знак больше, меньше, равно, не равно **(>, < , = , #**) и саму константу. Такая возможность очень удобна для остановки при условии достижения какой-то переменной определенного значения при этом счетчик прохождений через контрольную точку автоматически задается большим числом.

Несколько примеров.

Допустим, программа ломается после длительной работы и неизвестно где. Удалось установить, что управление обязательно проходило чрез метку M100.

Ставим контрольную точку командой K .M100 1000000 и запускаем программу. На экран выдается каждое прохождение через метку и оказывается, что через 10457 прохождений происходит крах. Тогда запускаем программу заново, устанавливаем контрольную точку, но уже директивой K .M100 10457 и дожидаемся остановки в отладчике. Теперь можно идти по шагам и уточнять место ошибки.

Пример установки контрольной точки по записи в переменную X значения пять: K .X W1 =5

Пример установки контрольной точки по записи в переменную X значения не равного пяти: K .X W1 #5

Команда **–K** снятия контрольной точки имеет вид: -K [«адрес»]

Если адрес не указан, снимаются все ранее установленные контрольные точки. Команда просмотра контрольной точки имеет вид: K [«адрес»]

Если адрес не указан, выводится список всех контрольных точек.

Информация о контрольных точках имеет следующий вид:

```
1/0   0023:0040B8BC .BB W4
10/0 0023:0040122D .M K0
```

здесь первое число — заданный счетчик, через дробь число раз, реально пройденных через эту точку, далее адрес контрольной точки и соответствующее ему символьное имя, затем тип контрольной точки (только для аппаратных точек) и, если было задано, условие сравнения с константой. Такая же информация выдается при прохождении через любую из контрольных точек.

Команда **G**. Запуск отлаживаемой программы

Команда имеет вид:

[-]G [=<начало>] [<1 контрольная точка>] [<2 контрольная точка>]

здесь «начало» — это адрес, с которого надо начать выполнение программы, если он не указан, выполнение начнется с текущего адреса в RIP, что практически всегда и нужно.

«контрольные точки» — это оперативные контрольные точки (см. предыдущий раздел). Если не установлены постоянные контрольные точки командой K и не указано оперативных точек в команде G, то программа будет выполняться без контроля отладчика (если конечно в самой программе нет вызовов исключения отладки INT 3). Таким образом, если вызвана программа с активацией отладчика и задана G без параметров, то программа выполнится как при обычном вызове и присутствие отладчика фактически никак не скажется.

В программе можно также установить несколько команд INT 3, (на PL/1 это обращение к оператору МАШ_КОД('CC'B4);) и каждая такая команда сработает как постоянная контрольная точка. Правда, эта контрольная точка никогда не снимется, и чтобы продолжить программу ее придется обойти, например, командой N

Примеры:

g =.m1 .m2

начать программу с метки m1 и остановиться, если дошли до m2;

g .m2

начать программу с текущего места (например, сначала) и остановиться, если дошли до m2;

g =.m1 .m2 .m3

начать программу с метки m1 и остановиться, если дошли до m2 или до m3;

Если при выполнении программы после команды G, она проходит через контрольные точки, установленные командой K, на экран выдается сообщение об этом, однако такие сообщения можно отменить, если перед буквой G задать знак «-» (минус).

Команда **T**. Пошаговое выполнение программы (трассировка)

Команда имеет вид: [-]T «число»

По команде T исполняется текущая команда процессора, и управление вернется в отладчик. Этот режим, называемый трассировкой, использует аппаратную возможность процессора вырабатывать сигнал прерывания после выполнения одной команды (исключение пошагового режима).

После каждого шага, отладчик выдает стандартную информацию на экран. Задав «число» можно выполнить указанное число команд без остановки после каждой, причем такое выполнение можно прервать посередине, нажав CTRL+C**.** Используя префикс «-», можно отменить выдачу стандартной информации на промежуточных шагах.

Команда **N**. Переход через INT 3

Команда имеет вид: N

Когда программа остановилась на команде INT 3, перейти к следующей команде командой трассировки не удастся. В этом случае следует изменить на 1 значение RIP, что и сделает отладчик по этой команде. Перепрыгнуть такой директивой через любую другую команду так не удастся, отладчик выдаст сообщение об ошибке.

Команда **P**. Пошаговое выполнение с пропуском подпрограмм

Команда имеет вид: [-]P «число»

Команда P без параметров аналогична команде T, за исключением того, что она выполняет за один шаг команды вызова подпрограмм (CALL) и «цепочечные» команды (типа REP MOVS). В этих случаях ставятся неявные контрольные точки. В остальных случаях команда действует как T.

Возможно, эта команда будет использоваться чаще всего, так как она позволяет пропускать служебные вызовы. Можно задать параметр «число», чтобы выполнить без остановки соответствующее число команд P, а использование префикса «-» отменит выдачу на экран стандартной информации на промежуточных шагах. Использование команды P может оказаться мощным средством поиска ошибочных подпрограмм. Дойдя до вызова очередной подпрограммы, можно посмотреть содержимое определенных данных до исполнения подпрограммы, затем командой P исполнить подпрограмму и смотреть в памяти результаты ее работы. Заметим, что команды -T «число» и -P «число» работают гораздо медленнее, чем, если бы просто непрерывно выполнилось «число» команд программы. Ведь в этом случае, после каждой команды программы происходит выход в отладчик, а затем опять вход в текущее место программы.

Команда **B**. Построчное выполнение с пропуском подпрограмм

Команда имеет вид B «номер строки» или [-]B «количество строк»

Кроме пошагового исполнения отладчик имеет возможность при отладке исполнять PL/1-программу по строке исходного текста. Для этого следует использовать параметр компиляции R (по этому ключу в исполняемый файл

занесется информация о строках), и тогда при отладке можно использовать команду B. Команда B без параметров выполнит команды программы, пока не изменится текущий номер строки исходного текста. Другими словами, выполнит одну строку исходного текста. Если же задать «номер строки» остановка произойдет, когда номер текущей строки совпадет с заданным, т.е. на первой команде заданной строки. Есть другая форма команды, с префиксом «-», здесь параметр «количество строк» задает не номер, а сколько строк следует выполнить до остановки. –B без параметра считается командой –B 1.

Команда **R**. Вывод информации, запись регистров и флагов

Команда имеет вид: R «имя регистра» или R «имя флага»

По команде R без параметров на экран выводится стандартная информация для текущего места в программе.

Если после R указать без пробела «имя регистра» или «имя флага», отладчик предложит изменить его значение.

В качестве имен регистров используются только две последних буквы их имени. Таким образом, допустимы только команды:

RAX RBX RCX RDX RSP RBP RSI RDI RIP

после указания одной из этих команд отладчик выведет полное значение регистра на экран и будет ожидать ввода. Если ввести символ точки, ввод прекратится, и значение не изменится. Если нажать «Enter», значение также не изменится, но отладчик выведет значение следующего по порядку регистра (от RBX до RIP) и опять будет ожидать ввода нового значения.

Если набрать новое значение и нажать «Enter», значение изменится и отладчик выведет значение следующего по порядку регистра (от RBX до RIP) и опять будет ожидать ввода нового значения. Таким образом, можно посмотреть и задать полные значения всех регистров, включая R8-R15.

Командой типа RAX «значение» можно сразу присвоить регистру RAX значение, не переходя в интерактивный режим ввода. Однако для регистров R8-R15 единственный способ изменения и просмотра — интерактивный ввод. Для ускорения, просмотр и изменение R8-R15 можно проводить командой RDI, та как следующий в списке по порядку регистр —– R8. В качестве значений для регистров могут быть и символические имена. По команде RIP меняется текущее место в программе, и теперь ее выполнение может быть продолжено с заданного места.

Например, можно повторить выполнение команд в программе, если указать RIP «адрес», где «адрес» — место, через которое уже прошло управление.

Просмотр и изменение флагов процессора производится командой R, за которой без пробела указывается буква флага, например, RC. Значения флажков меняются только интерактивно и только по одному. Они могут иметь значения только 0 или 1. Имена флагов:

Символ флага	Назначение флага
O	переполнение
D	направление
I	разрешение прерываний
T	пошаговая трассировка
S	отрицательный результат
Z	нулевой результат
A	дополнительный перенос
P	контроль четности
C	перенос

Команда **Z**. Просмотр регистров FPU

По команде Z на экран выводится текущее состояние рабочих и служебных регистров FPU. Если в регистры ST0-ST7 ничего не записывалось, они будут помечены как «пустые».

Например,

```
z
  CW=0360  SW=0000  TW=0000  IP=00000000  OP=00000000 DBE3/FNINIT
   ПУСТОЙ РЕГИСТР    ПУСТОЙ РЕГИСТР    ПУСТОЙ РЕГИСТР    ПУСТОЙ РЕГИСТР
   ПУСТОЙ РЕГИСТР    ПУСТОЙ РЕГИСТР    ПУСТОЙ РЕГИСТР    ПУСТОЙ РЕГИСТР
```

Кроме регистров ST0-ST7 выводится управляющее слово CW и слово текущего состояния SW, а также адрес последнего обращения к FPU и код и имя последней выполненной команды FPU.

Имеется также несколько разновидностей команды Z:

ZS для показа регистров XMM:

```
zs
X0= 0.0000000000E+000 0.0000000000E+000 X1= 0.0000000000E+000 0.0000000000E+000
X2= 0.0000000000E+000 0.0000000000E+000 X3= 0.0000000000E+000 0.0000000000E+000
X4= 0.0000000000E+000 0.0000000000E+000 X5= 0.0000000000E+000 0.0000000000E+000
X6= 0.0000000000E+000 0.0000000000E+000 X7= 0.0000000000E+000 0.0000000000E+000
```

ZM для показа регистров MMX в 16-ричном виде

```
zm
  00000000-00000000  00000000-00000000  00000000-00000000  00000000-00000000
  00000000-00000000  00000000-00000000  00000000-00000000  00000000-00000000
```

ZA для показа регистров 3DNow в «плавающем» виде

```
za
 0.00E+000 0.00E+000 0.00E+000 0.00E+000 0.00E+000 0.00E+000 0.00E+000 0.00E+000
```

```
0.00E+000 0.00E+000 0.00E+000 0.00E+000 0.00E+000 0.00E+000 0.00E+000 0.00E+000
```

Команда H. Просмотр имен и шестнадцатеричная арифметика

Команда имеет несколько разновидностей:

H «имя» или

H «адрес» или

HH или

H «шестнадцатеричное число» « шестнадцатеричное число»

Если для программы сформирован файл имен (с расширением .SYM), то команда H может работать с этим списком имен. H без параметров покажет весь список имен. Список выводится поэкранно и останавливается до нажатия любой клавиши. По клавише ESC выдача прекращается.

Если задать в качестве параметра «имя» (имя всегда должно начинаться с символа точка) отладчик ищет заданное имя в списке имен, доступных программе и в случае, если имя нашлось, выводит на экран его смещение, в соответствии с таблицей имен. Если для программы сформирован и файл .TRF, содержащий характеристики переменных в терминах PL/1, то кроме адреса и имени будут выведены и эти характеристики. Например:

```
h .deltadate
00415578 .DELTADATE     FLOAT(53)
```

Если задать в качестве параметра шестнадцатеричный «адрес» можно выяснить, какое имя располагается по заданному адресу:

```
h 415578
00415578 .DELTADATE     FLOAT(53)
```

Если такого имени не нашлось, отладчик покажет ближайшее к заданному адресу имя. Если это имя подпрограммы, то по выданному адресу можно легко сориентироваться, в каком месте программы остановился отладчик. Поскольку чаще всего необходимо определиться в текущем месте программы, то можно выполнить команду: H EIP или в более простой форме без параметров командой HH:

```
h eip
Внутри 00401200 .SIG_PAR        PROC
hh
Внутри 00401200 .SIG_PAR        PROC
```

Эта же команда H «число» «число» используется и для простейшей шестнадцатеричной арифметики, чтобы простые действия с адресами проводить здесь же, не вызывая приложение типа «калькулятор».

Для шестнадцатеричной арифметики задается два шестнадцатеричных числа через пробел в качестве параметров.

В результате выводится строка результатов, которая содержит десятичное представление чисел (со значком #) и далее результаты сложения, вычитания, умножения, деления этих двух чисел. Результат деления состоит из двух чисел – частное и остаток:

```
h 100 20
#256 #32 +00000120 -000000E0 *0000000000002000 /00000008:00000000
```

Команда **U**. Вывод команд в виде строк ассемблера

Команда имеет ряд разновидностей:

[-]U [«начало»] [«конец»] или
[-]U [«начало» +»число»] или
U/ или
U*«адрес»

По команде U на экран выводится указанная область памяти в виде команд ассемблера. Если «конец» или «число» не заданы, выводится 12 строк. Если «начало» не задано, выводится с текущего места в программе. Если указан знак плюс, то перед ним должен быть пробел, а после — нет.

Примеры:

```
U 401200
U .m1 +100
U .НАЧАЛО
```

В каждой строке выводится метка, если есть, текущий адрес команды, ее шестнадцатеричный код, собственно сама команда, символьное имя, если операнд совпал с адресом какой-либо переменной таблицы имен:

```
00401344 E8FC6C0000                CALL   00408045 .?SIOOP
00401349 E8DE720000                CALL   0040862C .?GNVOC
0040134E 7917                      JNS    00401367
00401350 F9                        STC
00401351 E8E59B0000                CALL   0040AF3B .?QCFOP
00401356 BBD84B4100                MOV    EBX,00414BD8
0040135B DD0424                    FLD64  [RSP]
0040135E DC0B                      FMUL64 [RBX]
00401360 58                        POP    RAX
00401361 DD1D51420100              FST64P [004155B8] .EPS
00401367 E8C0720000                CALL   0040862C .?GNVOC
0040136C 790D                      JNS    0040137B
```

Вывод шестнадцатеричного кода можно отменить (или обратно установить) командой U/. Вывод имен можно отменить, если использовать префикс "-". Иногда требуется просмотр нескольких команд не «вниз» от текущего места, а вверх. Это можно сделать с помощью команды U*:

```
u*401367
00401347 0000                      ADD    [RAX],AL
00401349 E8DE720000                CALL   0040862C .?GNVOC
0040134E 7917                      JNS    00401367
00401350 F9                        STC
00401351 E8E59B0000                CALL   0040AF3B .?QCFOP
00401356 BBD84B4100                MOV    EBX,00414BD8
```

```
0040135B DD0424                    FLD64  [RSP]
0040135E DC0B                      FMUL64 [RBX]
00401360 58                        POP    RAX
00401361 DD1D51420100              FST64P [004155B8] .EPS
00401367*E8C0720000                CALL   0040862C .?GNVOC
0040136C 790D                      JNS    0040137B
```

При этом выводятся команды с адреса на 64 байта меньшего, а команда по заданному адресу будет помечена звездочкой. Однако может так случиться, что команды выше будут выведены неправильно, поскольку их начала неизвестны. В примере самая верхняя команда по адресу 401347 декодирована с середины и неправильно. В этом случае надо еще увеличить число байт «вверх»:

```
u*401367-20
00401327 41008F07E8C56E            ADD    [R15]+6EC5E807,CL
0040132E 0000                      ADD    [RAX],AL
00401330 45BFC7025349              MOV    R15D,495302C7
00401336 BB30C64100                MOV    EBX,0041C630
0040133B B994134000                MOV    ECX,00401394
00401340 66BA9006                  B32:   MOV    DX,0690
00401344 E8FC6C0000                CALL   00408045 .?SIOOP
00401349 E8DE720000                CALL   0040862C .?GNVOC
0040134E 7917                      JNS    00401367
00401350 F9                        STC
```

Теперь команды по адресу 401327 декодированы неправильно, но зато команда по адресу 401344 (а не 401347) теперь декодирована правильно.

Команда **E**. Изменение содержимого памяти

Команда имеет вид: E «начало» или EW «начало».

Команда позволяет изменить содержимое памяти, начиная с указанного адреса побайтно. После ввода команды, на экран выдается содержимое очередного байта и ожидается ввод. Если просто нажать «Enter», содержимое байта не изменится и произойдет переход к следующему. Окончание ввода — символ «точка», например байты B8 и 92 не меняются, а байт 88 нужно изменить на 89:

```
e 401200
 B8
 88 89
 92 .
```

Память можно изменять не байтами, а словами, в этом случае, используется EW.

Команда **F**. Заполнение памяти

Команда имеет несколько форм:

F «начало» «конец» «значение» или

F «начало» +«число» «значение» или

FW «начало» «конец» «значение» или

FW «начало» +«число» «значение»

Иногда требуется участок памяти заполнить одинаковым значением (например, нулями). В этом случае используется команда F или FW, если память должна быть заполнена словами. Необходимо, чтобы «значение» было числом не более FF для заполнения байтами и FFFF для заполнения словами. Перед плюсом должен быть хотя бы один пробел, иначе он воспримется как выражение для начального адреса.

Примеры:

```
f 401200 401400 00
f 401200 +200 #22
fw 401200 401400 5566
```

Команда **S**. Поиск заданного значения в памяти

Команда имеет вид:

S «начало» «конец» «значение» или

S «начало» +«число» «значение»

По команде S отладчик ищет заданное значение в указанном участке памяти. Значение должно быть или байтом, цепочкой байт или символьной строкой. Перед плюсом должен быть хотя бы один пробел, иначе он воспримется как выражение для начального адреса. Цепочку байт следует вводить через пробел, начиная с младшего, аналогично тому, как они выглядят на экране по команде D. Символьную строку следует начать с двойной кавычки. Если значение найдено, на экран выдается его адрес в памяти, если не найдено — ничего и не выдается.

Примеры:

```
s 400100 400200 #22
s 401100 402200 ac a0 ae a0 ac
s 400100 +80  "НАЧАЛО
```

Команда **C**. Сравнение участков памяти

Команда имеет вид:

C «начало1» «конец» «начало2» или

C «начало1» +«число» «начало2»

По команде C отладчик сравнит между собой два заданных участка памяти побайтно. Для не совпавших байт выдается адрес и оба значения байта. Если участки совпали — ничего не выдается. Перед плюсом должен быть хотя бы один пробел, иначе он воспримется как выражение для начального адреса.

Команда **M**. Перенос участков памяти

Команда имеет вид:

M «начало1» «конец» «начало2» или
M «начало1» +«число» «начало2»

По команде M отладчик пересылает один участок памяти в другой. Перед плюсом должен быть хотя бы один пробел, иначе он воспримется как выражение для начального адреса.

Пример:

```
m 401200 +100 402000
```

теперь сравнение дает полное совпадение

```
c 401200 +100 402000
```

Команда **L** перехвата всех ON-операторов в программе

В PL/1-программе могут быть установлены различные ON/КОГДА-операторы. При возникновении указанных в этих операторах состояний, управление будет передаваться на них. Это затрудняет пошаговую отладку. Например, директивой P отладчика выполняли системную подпрограмму открытия файла. Но если указанный файл не найден, и управление передалось на заданный обработчик состояния НЕ_НАЙДЕН_ФАЙЛ, то установленная отладчиком контрольная точка после подпрограммы не сработает, программа «отцепится» от пошаговой трассировки и начнет выполняться без участия отладчика. Это может ввести в заблуждение программиста, если он, например, искал, какая подпрограмма выдает ошибку, поскольку внешне именно при выполнении этой подпрограммы и выдалось сообщение на экран. На самом же деле, здесь программа только «отцепилась» от отладчика и выполнялась дальше и где-то далее произошла ошибка.

Чтобы избежать таких ситуаций, предусмотрена команда L без параметров, которая устанавливает контрольную точку внутри системной подпрограммы реакции на ситуации с именем «?ON». Через эту единственную точку начинают выполняться все обработчики состояний, и программист не теряет контроля над программой со стороны отладчика. Отменить такой перехват можно командой –K, поскольку это обычная контрольная точка.

Команда **V**. Информация о процессоре

По команде V без параметров можно получить на экране информацию о процессоре, доступную по команде процессора CPUID.

Команда **A**. Показ кириллицы Windows

Данная вспомогательная команда влияет только когда командой D выводится участок памяти как строки кода, и они же, как текст. Поскольку

обычно в консольном окне используется «кириллица DOS», а в окнах Windows – «кириллица Windows», данная команда преобразует в выводе отладчика на экран все в «кириллицу DOS», например:

```
d 401000 +#15
002B:00401000 8F CF 00 00 00 00 00 00-00 00 00 00 00 00 00 00 П╧..............
a
Font Win
d 401000 +#15
002B:00401000 8F CF 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ПП..............
```

После команды A и код 8F и код CF отображаются на экране как «П».

Чтобы упростить ввод одинаковых последовательностей команд существует возможность создания макрокоманд в отладчике.

Макрокоманда — это последовательность нескольких команд отладчика, объединенных под некоторым именем, которую можно вызвать, как обычную команду. Имена макрокоманд могут совпадать с именами обычных команд, поскольку вызов макрокоманд производится с помощью знака равенства. При этом вся заданная последовательность выполнится в автоматическом режиме. Признаком задания нового макро является двоеточие, признаком исполнения — знак равенства.

Новая макрокоманда определяется следующим образом:

— вводится имя новой макрокоманды с символом двоеточие;

— затем набираются команды отладчика по одной на строке;

— пустая строка является концом макрокоманды.

Пример ввода макрокоманды с именем T:

```
:t
d 400100 400110 «Enter»
d 400200 400210 «Enter»
«Enter»
```

Вызов макрокоманды производится по ее имени, перед которым ставится знак равенства. Таким образом, если теперь ввести в отладчике новую команду =T, в автоматическом режиме выполнится сначала d 400100 400110, а затем d 400200 400210.

Проверь себя

1. Что такое контрольные точки, каких типов они бывают? Можно ли сказать, что контрольные точки — это основа работы с отладчиком?

2. При пошаговом исполнении программы (трассировки) дошли до адреса 404300, где стоял вызов процедуры. Попытка исполнения процедуры командой P привела к краху программы. Как после повторного запуска сразу оказаться в этой же точке программы, не повторяя длительной трассировки?

3. Как после краха программы и выхода в отладчик определить, какие процедуры были вызваны и не завершили работу? Подсказка: адреса возврата из незавершенных процедур обычно остаются в стеке, на вершину которого указывает регистр RSP (директива просмотра D ESP).

4. Программа после длительной работы терпит крах. При работе программы управление обязательно и многократно проходило через метки M1, M2, M3. Как с помощью отладчика установить, через какую из трех этих меток управление проходило последним перед крахом?

5. После выхода в отладчик из-за ошибки анализом установлено, что переменная *x* длиной в байт получила недопустимое значение FFH. Как при повторном запуске программы с помощью отладчика выяснить, в каком месте программы этой переменной присваивается недопустимое значение? Можно ли отладчиком в этом месте исправить значение переменной на ноль?

6. Какой директивой отладчика можно перехватывать управление в случаях возникновения состояний, для которых в программе предусмотрена своя обработка (ON/КОГДА-операторы)? Что может произойти при отладке, если такой перехват не устанавливать?

13. ПРОЧИЕ ВОЗМОЖНОСТИ ЯЗЫКА PL/1

13.1. Параллельное выполнение фрагментов программы

Обычно (и во всех приведенных выше примерах) операторы программы на языке PL/1 выполняются последовательно. Например, если в программе стоит вызов процедуры, то управление переходит внутрь этой процедуры и возвращается на следующий оператор (или в другую точку) только после выполнения процедуры. В любом случае, точка, где находится управление в программе, остается всегда единственной. Однако современные процессоры имеют несколько *ядер* — устройств, имеющих собственные точки управления и собственное состояние, т.е. собственный набор регистров. Как правило, при этом они разделяют общую память. Таким образом, возможен вызов процедуры на другом ядре, когда точки управления как бы раздваиваются: одна точка переходит на первый оператор процедуры, а вторая — на оператор, стоящий сразу после оператора вызова. Далее эти две точки управления (в терминах Windows два *потока*) начинают работать физически одновременно.

К процедуре, запущенной на выполнение *параллельно*, предъявляются определенные требования. Во-первых, она не может быть функцией и возвращать значение в выражение. Во-вторых, по достижении своего последнего оператора (или оператора RETURN/ВОЗВРАТ) управление больше не попадет на оператор, стоящий за оператором параллельного вызова. Если при вызове точки управления как бы раздваивались, то теперь одна точка, которая была внутри процедуры, просто исчезает. Можно вновь запустить ту же или другую процедуру, тогда число точек управления (потоков) опять увеличится.

В языке PL/1 имеются некоторые средства поддержки параллельного исполнения. В рассматриваемой версии языка они сводятся:

— к специальной форме оператора параллельного вызова процедуры в виде ***p* TASK** или ***p* ПРОЦЕСС**, где *p* — «обычный» оператор вызова процедуры CALL/ВЫЗОВ с параметрами или без;

— к специальной форме оператора останова параллельного вызова процедуры в виде ***p* STOP** или ***p* СТОП**, где *p* — «обычный» оператор вызова процедуры CALL/ВЫЗОВ с параметрами или без (аргументы формально должны быть указаны, но не используются);

— к оператору задержки **DELAY/ЗАСНУТЬ** в виде встроенной функции, параметром которой является целочисленное выражение — число миллисекунд, на которые должна «остановиться» выполняемая процедура;

— к оператору ожидания **WAIT/ЖДАТЬ** с параметром в виде битово-строчной переменной или выражения с битово-строчным значением и длиной в один бит (подразумевается, что это выражение может измениться в результате параллельной работы каких-либо процедур).

Таким образом, если в программе последовательный вызов процедуры осуществляется оператором call $f(x_1, x_2, \ldots, x_n)$; или просто $f(x_1, x_2, \ldots, x_n)$; то параллельный запуск осуществляется оператором call $f(x_1, x_2, \ldots, x_n)$ task; или вызов $f(x_1, x_2, \ldots, x_n)$ процесс; а остановка и снятие этой процедуры — оператором call $f(x_1, x_2, \ldots, x_n)$ stop; или вызов $f(x_1, x_2, \ldots, x_n)$ стоп; причем, параллельно запущенная процедура f снимется и перестанет работать, когда она дойдет до оператора RETURN/ВОЗВРАТ или до своего последнего оператора или какая-либо процедура (или даже сама процедура f) выполнит оператор вызова с ключевым словом STOP/СТОП.

Рассмотрим следующий пример. Пусть в программе имеются две процедуры, выполняющие некоторые вычисления.

```
CONCURRENT_TEST:ПРОЦ ГЛАВНАЯ;
ОПС
(F1,F2)        ВЕЩ(53),     // ПАРАЛЛЕЛНО ВЫЧИСЛЯЕМЫЕ РЕЗУЛЬТАТЫ ФУНКЦИЙ
(ФЛАГ1,ФЛАГ2) БИТ;         // ПРИЗНАКИ, ЧТО ВЫЧИСЛЕНИЯ ОКОНЧЕНЫ

//---------------- ОПИСАНИЕ ПЕРВОЙ ПАРАЛЛЕЛЬНОЙ ЗАДАЧИ ----------------
CONCURRENT1:ПРОЦ(X1,X2);
ОПС
I       ТОЧНОЕ(31),        // ВНУТРЕННИЙ СЧЕТЧИК
(X1,X2) ВЕЩ(53);           // ВХОДНОЙ И ВЫХОДНОЙ ПАРАМЕТРЫ
X2=0;
//---- ДЛЯ ПРИМЕРА СЧИТАЕМ НЕЧТО ВРОДЕ РЯДА ФУРЬЕ ----
ЦИКЛ I=1 ДО 100_000_000;
   X2+=COS(ВЕЩ(2*I-1,53))*X1/(2*I-1)/(2*I-1);
КОНЕЦ I;
//---- ПРИЗНАК ОКОНЧАНИЯ РАСЧЕТА, ОТВЕТ В X2 ----
ФЛАГ1='1'Б;
КОНЕЦ CONCURRENT1;

//---------------- ОПИСАНИЕ ВТОРОЙ ПАРАЛЛЕЛЬНОЙ ЗАДАЧИ ----------------
CONCURRENT2:ПРОЦ(X1,X2);
ОПС
I       ТОЧНОЕ(31),        // ВНУТРЕННИЙ СЧЕТЧИК
(X1,X2) ВЕЩ(53);           // ВХОДНОЙ И ВЫХОДНОЙ ПАРАМЕТРЫ
X2=0;
//---- ДЛЯ ПРИМЕРА СЧИТАЕМ НЕЧТО ВРОДЕ РЯДА ФУРЬЕ ----
ЦИКЛ I=1 ДО 100_000_000;
   X2+=SIN(FLOAT(2*I-1,53))*X1/(2*I-1)/(2*I-1);
КОНЕЦ I;
```

```
//---- ПРИЗНАК ОКОНЧАНИЯ РАСЧЕТА, ОТВЕТ В Х2 ----
ФЛАГ2='1'Б;
КОНЕЦ CONCURRENT2;
//---------------- СОБСТВЕННО ВЫПОЛНЕНИЕ РАСЧЕТА --------------------
//---- ЗАПУСК РАСЧЕТА ----
ПИСАТЬ С_НОВОЙ В_ВИДЕ('НАЧАЛО СЧЕТА',TIME);

//---- ЗАПУСК В ПАРАЛЛЕЛЬНОМ РЕЖИМЕ ----
CONCURRENT1(5E0,F1) ПРОЦЕСС;
CONCURRENT2(5E0,F2) ПРОЦЕСС;

//---- ЖДЕМ ОКОНЧАНИЕ РАСЧЕТА ----
ЖДАТЬ((ФЛАГ1='1'Б) И (ФЛАГ2='1'Б));

//---- ВЫДАЕМ РЕЗУЛЬТАТЫ РАСЧЕТА -----
ПИСАТЬ С_НОВОЙ В_ВИДЕ('КОНЕЦ  СЧЕТА',TIME);
ПИСАТЬ С_НОВОЙ С_ИМЕНАМИ(F1,F2);
КОНЕЦ CONCURRENT_TEST;
```

в этой программе параллельно запускаются две процедуры, выполняющие сходные расчеты. Каждая из процедур сигнализирует об окончании своей работы установкой битово-строчной переменной в единицу. Главная программа (или главный поток) ждет, пока обе переменные не станут единичными, после чего печатает результат. Если в тексте программы убрать описатели ПРОЦЕСС, то результат вычислений будет такой же, но подпрограммы выполнятся последовательно, а не параллельно.

Запуск на конкретном компьютере в параллельном режиме дал результат

```
НАЧАЛО СЧЕТА 17:44:45
КОНЕЦ  СЧЕТА 17:44:57
F1=  2.24151193369346E+000 F2=  4.16086309822767E+000
```

Запуск в последовательном режиме дал результат

```
НАЧАЛО СЧЕТА 17:45:37
КОНЕЦ  СЧЕТА 17:45:52
F1=  2.24151193369346E+000 F2=  4.16086309822767E+000
```

в параллельном режиме расчет выполнен за 12 секунд, а в последовательном — за 15 секунд. Хотя разница в данном случае невелика (вероятно, члены ряда при вычислениях быстро превращаются в ноль), видно, что точно такие же вычисления в параллельном режиме выполнились быстрее.

Оператор ожидания WAIT/ЖДАТЬ реализован как цикл DO WHILE/ЦИКЛ ПОКА с телом из одного оператора ЗАСНУТЬ(0); так как задержка на ноль миллисекунд просто сообщает операционной системе, что данная программа находится в режиме ожидания и все время до конца текущего кванта, выделенного данной программе может быть использовано для других программ.

При организации взаимодействия параллельно работающих процедур часто используют такие простые элементы как *семафоры*. Семафор — это

просто общая переменная, обычно принимающая значение 0 и 1. Каждый семафор указывает на то, свободен или занят некоторый ресурс (например, обращение к файлу) для параллельно работающих процедур. Предполагается, что первая из процедур, захватившая этот ресурс, должна установить семафор в 1, а после освобождения ресурса — в ноль. Остальные процедуры, которым потребовался этот же ресурс, должны ожидать нулевого значения семафора. Чтобы не было коллизий при установке семафора, его значение нельзя менять обычным оператором присваивания. Вместо этого используется локальная переменная в каждой процедуре, которая также принимает значения 0 и 1, например, следующим образом:

```
опс S1 бит(1) общее,          // семафор некоторого ресурса
    S0 бит(1);                // локальная переменная
S0='1'Б;
S1 <=> S0;
если НЕ S0 тогда …            // используем захваченный ресурс
        иначе ждать(S1);      // ждем освобождения ресурса
```

установка семафора S1 производится неделимой операцией обмена <=>, если семафор S1 был нулевой — туда запишется единичное значение от S0, а в переменную S0 — признак, что ресурс свободен. Если же семафор S1 был уже единичный, его состояние не изменится, а в локальную переменную S0 запишется признак занятости ресурса.

Проверь себя

1. Какие фрагменты программ в языке PL/1 можно выполнять в параллельном режиме?
2. Чем текстуально отличается запуск в параллельном режиме от запуска в последовательном режиме?
3. Пусть процедура, запущенная в параллельном режиме, имеет множество вложенных циклов и в самом внутреннем цикле нашла нужный элемент? Как ей проще всего прекратить свою работу? Можно ли в этом случае использовать оператор STOP/СТОП?
4. Может ли процедура, запустившая другую процедуру в параллельном режиме, принудительно завершить ее работу, не дожидаясь окончания ее работы?
5. Может ли программист смоделировать работу оператора WAIT/ЖДАТЬ другими средствами?
6. Что такое семафор, и каким оператором PL/1 он изменяется?

13.2. Массивы с изменяемыми границами

Обычно реализация массивов с границами, изменяемыми при выполнении программы, состоит в размещении специальной информации о таком массиве («паспорте» массива), часто располагаемой перед данными собственно массива в памяти. Таким образом, при выполнении программы доступ к элементам массива получается двухступенчатым: сначала достается информация из «паспорта» о границах, затем уже собственно формируется доступ. Если же границы массива заданы константами — «паспорта» не требуется, а доступ к элементам производится сразу, при этом границы-константы превращаются в некоторые *смещения-константы* в кодах программы.

В рассматриваемой версии языка не предусмотрено создание массивов с изменяемыми при работе программы границами. Однако в данной реализации такие массивы класса памяти CONTROLLED/СТЕК все-таки возможны, хотя для простоты компилятора, требуется некоторая дополнительная подготовка. Это позволяет генерировать всегда один и тот же код со смещениями-константами, а затем, в процессе выполнения программы, можно корректировать эти константы в зависимости от требуемых границ с помощью специальной встроенной функции «ретрансляции» **?RET**, которая уже описана в самом верхнем блоке как:

ОПС ?RET ДЛЯ_ВЫЗОВА(УКАЗ) ОБЩЕЕ;

Имеется также встроенный массив **?INDEX**, уже описанный в верхнем блоке как: ОПС ?INDEX(15,2) ТОЧНОЕ(31) ОБЩЕЕ;

Первая размерность этого массива 1:15, поскольку это максимально допустимое число размерностей массива в компиляторе. Служебная подпрограмма «ретрансляции», т.е. корректировки констант в командах, использует текущие значения массива ?index. При запуске программы все значения границ в массиве ?index по умолчанию заданы единичными.

Передача новых значений границ через внешний массив ?index, а не через входные параметры подпрограммы ?ret сделана для большей гибкости их использования. Например, в этом случае не требуется повторять задания границ для массивов одинаковых размеров, не требуется перечислять нижние границы каждой размерности, если они всегда остаются единичными и т.п.

В случае какой-либо ошибки корректировки констант, подпрограмма ?ret не возвращает код ошибки, но инициирует состояние ошибки, поскольку далее обращение к массиву, указанному как входной параметр, даст непредсказуемые результаты.

«Динамические» границы, обозначаемые «*», могут быть только старшими размерностями и следовать подряд. При этом такие границы могут быть не только у массивов однородных элементов, но и у массивов структур. Младшие размерности при этом могут оставаться константами, например:

```
опс // структура-массив с изменяемыми границами
1 S(*,*)        стек,
 2 X1           текст(100) рд,
 2 Y1 (-1:25) вещ,
 2 Z1 (100)    точное(31);
```

Для передачи в подпрограммы массивов с изменяемыми границами как параметров, учитывается тот факт, что такие массивы всегда неявно используют указатели. Но поскольку это служебные указатели, создаваемые компилятором, напрямую использовать их имена нельзя. Обращение к указателю без явного использования его имени возможно в языке PL/1 в случае применения встроенной функции ADDR/АДРЕС, например:

```
опс x(100) вещ основа(p1),
    (p1, p2) указ;
p2=адрес(x);          // это эквивалентно p2=p1;
```

Таким образом, если нужно передавать как параметры массивы с «динамическими» границами, достаточно передать указатели на них с помощью ADDR/АДРЕС, без явного использования имен служебных указателей: вызов умножение_матриц(адрес(a), адрес(b), адрес(c), m, n, q);
и тогда описание параметров таких подпрограмм становится единообразным, не зависящим от значений границ самих массивов:

```
опс умножение_матриц для_вызова(указ, указ, указ, точное(31), точное(31), точное(31));
```

Такой подход позволяет передавать «динамические» массивы в процедуры, но не позволяет «принимать» их внутри подпрограмм, поскольку тогда нужно присваивать указатели-параметры объектам класса «управляемой» памяти. Поэтому в компиляторе дополнительно разрешено использовать и в левой части присваивания встроенную функцию ADDR/АДРЕС:

```
опс x(*) вещ стек,
      p1   указ;
адрес(x)=p1; // эквивалентно оператору «служебный указатель на x»=p1;
```

Рассмотрим пример решения типовой задачи умножения матриц, как массивов с заранее неизвестными границами.«Динамические» массивы-матрицы создаются оператором ДАТЬ_ПАМЯТЬ и передаются (неявно указателями) в универсальную процедуру умножения матриц. Корректировка

констант в исполняемом коде производится как для фактических параметров A1, B1, C1, так и для формальных A, B, C. Кроме этого, формальным параметрам присваиваются фактические значения с помощью разрешения использовать функцию АДРЕС в левой части присваивания. Универсальная процедура умножения матриц может находиться в отдельном модуле и транслироваться автономно.

```
TEST:ПРОЦ ГЛАВНАЯ;
ОПС (A1,B1,C1)(*,*) ВЕЩ СТЕК;       // ДИНАМИЧЕСКИЕ МАТРИЦЫ
ОПС (M1,N1,Q1)      ТОЧНОЕ(31);     // ЗАДАВАЕМЫЕ ГРАНИЦЫ
ОПС (I,J)           ТОЧНОЕ(31);     // РАБОЧИЕ ПЕРЕМЕННЫЕ
//---- ДЛЯ ТЕСТА ЗАДАЕМ НЕКОТОРЫЕ ЗНАЧЕНИЯ ГРАНИЦ ----
M1=5; N1=4; Q1=3;
//---- КОРРЕКТИРУЕМ КОНСТАНТЫ A1(M1,N1), B1(N1,Q1), C1(M1,Q1) ----
?INDEX (1,2)=M1; ?INDEX(2,2)=N1;
?RET(АДРЕС(A1));                    // ИЗМЕНЯЕМ КОМАНДЫ ДЛЯ A1
?INDEX (1,2)=N1; ?INDEX(2,2)=Q1;
?RET(АДРЕС(B1));                    // ИЗМЕНЯЕМ КОМАНДЫ ДЛЯ B1
?INDEX (1,2)=M1;
?RET(АДРЕС(C1));                    // ИЗМЕНЯЕМ КОМАНДЫ ДЛЯ C1
//---- СОЗДАЕМ МАТРИЦЫ A1(M1,N1), B1(N1,Q1) И C1(M1,Q1) ----
ДАТЬ_ПАМЯТЬ A1,B1,C1;
//---- ДЛЯ ТЕСТА ЗАПОЛНЯЕМ ИХ НЕКОТОРЫМИ ЗНАЧЕНИЯМИ ----
ЦИКЛ I=1 ДО M1; ЦИКЛ J=1 ДО N1; A1(I,J)=I+J; КОНЕЦ J; КОНЕЦ I;
ЦИКЛ I=1 ДО N1; ЦИКЛ J=1 ДО Q1; B1(I,J)=I-J; КОНЕЦ J; КОНЕЦ I;
//---- УМНОЖЕНИЕ МАТРИЦ A1 И B1, РЕЗУЛЬТАТ - МАТРИЦА C1 ----
УМНОЖЕНИЕ_МАТРИЦ(ADDR (A1),ADDR(B1),ADDR(C1),M1,N1,Q1);
//---- ВЫДАЕМ ПОЛУЧЕННЫЙ РЕЗУЛЬТАТ ----
ПИСАТЬ С_НОВОЙ С_ИМЕНАМИ(C1);
ВЕРНУТЬ_ПАМЯТЬ A1,B1,C1;

//======== УМНОЖЕНИЕ МАТРИЦ ЗАДАННОГО РАЗМЕРА ========
УМНОЖЕНИЕ_МАТРИЦ:ПРОЦ(P1,P2,P3,M,N,Q);
//---- ВХОД A(M,N) И B(N,Q), ОТВЕТ - МАТРИЦА C(M,Q) ----
ОПС (P1,P2,P3)   УКАЗ;          // УКАЗАТЕЛИ НА МАТРИЦЫ
ОПС (M,N,Q)      ТОЧНОЕ(31);    // ЗАДАННЫЕ ГРАНИЦЫ
ОПС (A,B,C)(*,*) ВЕЩ СТЕК;      // ДИНАМИЧЕСКИЕ МАТРИЦЫ
ОПС (I,J,K)      ТОЧНОЕ(31);    // РАБОЧИЕ ПЕРЕМЕННЫЕ
//---- ОПЕРАТОРЫ ПРИСВАИВАНИЯ УКАЗАТЕЛЕЙ ----
АДРЕС(A)=P1;     // АДРЕС ДЛЯ МАССИВА A
АДРЕС(B)=P2;     // АДРЕС ДЛЯ МАССИВА B
АДРЕС(C)=P3;     // АДРЕС ДЛЯ МАССИВА C
//---- КОРРЕКТИРУЕМ КОНСТАНТЫ МАТРИЦ A(M,N), B(N,Q), C(M,Q) ----
?INDEX (1,2)=M; ?INDEX(2,2)=N;
?RET(АДРЕС(A));                     // ИЗМЕНЯЕМ КОМАНДЫ ДЛЯ A
?INDEX (1,2)=N; ?INDEX(2,2)=Q;
?RET(АДРЕС(B));                     // ИЗМЕНЯЕМ КОМАНДЫ ДЛЯ B
?INDEX (1,2)=M;
?RET(АДРЕС(C));                     // ИЗМЕНЯЕМ КОМАНДЫ ДЛЯ C
//---- СОБСТВЕННО УМНОЖЕНИЕ МАТРИЦ ----
```

```
    ЦИКЛ I=1 ДО M;
       ЦИКЛ J=1 ДО Q;
            C(I,J)=0;
            ЦИКЛ K=1 ДО N;
                 C(I,J)+=A(I,K)*B(K,J);
            КОНЕЦ K;
       КОНЕЦ J;
    КОНЕЦ I;
    КОНЕЦ УМНОЖЕНИЕ_МАТРИЦ;
    КОНЕЦ TEST;
```

Результат запуска программы:

```
C1(1,1)=  2.600000E+01 C1(1,2)=  1.200000E+01 C1(1,3)= -2.000000E+00
C1(2,1)=  3.200000E+01 C1(2,2)=  1.400000E+01 C1(2,3)= -4.000000E+00
C1(3,1)=  3.800000E+01 C1(3,2)=  1.600000E+01 C1(3,3)= -6.000000E+00
C1(4,1)=  4.400000E+01 C1(4,2)=  1.800000E+01 C1(4,3)= -8.000000E+00
C1(5,1)=  5.000000E+01 C1(5,2)=  2.000000E+01 C1(5,3)= -1.000000E+01
```

Проверь себя

1. Как обычно реализуются массивы с изменяемыми границами?
2. В чем преимущество эффективности кода программы для массивов с границами-константами по сравнению с изменяемыми во время исполнения программы, заранее неизвестными границами?
3. Нужно ли описывать массив ?INDEX и процедуру ?RET ?
4. Используя приведенный выше пример, разработайте процедуру решения стационарного уравнения теплопроводности методом Либмана: пусть имеется квадратная матрица T размером NxN элементов типа вещ(53), у которой первоначально все элементы нулевые, а первая строка и первый столбец единичные. Требуется провести 10000 итераций расчета и напечатать получившуюся матрицу. Алгоритм расчета:

```
DTM=0;
ЦИКЛ I=2 ДО N-1;
     I1=I-1; I2=I+1;
     ЦИКЛ J=2 ДО N-1;
          T1=(T(I1,J)+T(I2,J)+T(I,J+1)+T(I,J-1))/4;
          DTM=MAX(ABS(T1-T(I,J)),DTM);
          T(I,J)=T1;
     КОНЕЦ J;
КОНЕЦ I;
```

Проведите расчет для N=5000.

13.3. Расширение типизации «физическими» типами

При решении ряда задач некоторым операторам, например, присваивания можно сопоставить уравнения, где элементы справа и слева имеют понятие «размерности» в терминах международной системы единиц СИ. Если ввести в

язык понятие «физического» типа и приписывать такой тип переменным программы дополнительно к уже имеющимся типам, то на уже этапе компиляции можно проверить правильность записи ряда операторов с точки зрения теории размерности. Это реализовано в рассматриваемой версии языка.

Поскольку в СИ независимыми являются лишь семь физических величин:
— длина **L** (единица — метр),
— масса **M** (единица — килограмм),
— время **T** (единица — секунда),
— термодинамическая температура **Θ** (единица — Кельвин),
— сила электрического тока **I** (единица — ампер),
— сила света **J** (единица — кандела),
— количество вещества **N** (единица — моль),
то в программе любой переменной, которой приписан физический смысл, может быть сопоставлен «физический» же тип в виде размерности из степенного одночлена, выраженного через основные единицы: $[x]=L^l\ M^m\ T^t\ \Theta^\theta\ I^i\ J^j\ N^n$.

Таким образом, в программе переменным может быть назначено произвольное число видов «физического» типа, каждый из которых представляется при трансляции одинаково — степенным одночленом, имеющим возможно разные показатели степеней, в том числе нулевые. Если все показатели степеней нулевые — объект не имеет физической размерности.

На практике величины, имеющие физический смысл, часто задаются в производных единицах измерения, отличающихся от базовых единиц масштабным коэффициентом. Например, длины — в км, площади — в Га и т.д. Поэтому целесообразно ввести в представление «физических» типов ещё и масштабный коэффициент в виде одного числа в формате IEEE-754.

Наличие такого коэффициента позволит компилятору автоматически учитывать масштабы при генерировании операций для выражений. Например, если переменным X1, X2, X3 приписаны типы соответственно [км], [см] и [мм], то при вычислении выражения X1=X2+X3, компилятор должен автоматически добавить операции умножения X2 на 0.01 и X3 на 0.001, а затем результат сложения перед присваиванием X1 ещё разделить на 1000.

Наличие приближенного масштабного коэффициента дает важное следствие — «физические» типы могут быть тоже только у приближенных переменных, т.е. только у переменных в формате FLOAT/ВЕЩ. При этом автоматическая подстановка масштабных коэффициентов при трансляции выражений означает, что все расчеты в программе будут производиться в базовых величинах СИ, как в приведенном выше примере — в метрах. Это также означает, что масштабные коэффициенты в «физическом» типе могут

быть только у переменных, но не у выражений, где масштабы уже учтены при генерации операций загрузки переменных этого выражения.

Хотя в СИ имеется лишь семь базовых величин, на практике используются ещё две вспомогательные величины — *плоский угол* (единица — радиан) и *телесный угол* (единица — стерадиан). Здесь наблюдается некоторое противоречие с точки зрения формального определения типа. В самом деле, радиан и стерадиан — по определению безразмерные величины и могут быть использованы или не использованы в выражениях для других производных единиц СИ. Однако в некоторых случаях они ведут себя как обычные (размерные) величины. Например, имеется встроенная функция SIN, входным параметром которой должно быть выражение в радианах и функция SIND, входным параметром которой должно быть выражение в градусах. Переменная с «физическим» типом «градус» отличается от переменной с «физическим» типом «радиан» только масштабным коэффициентом π/180.
Таким образом, целесообразно и радианы и стерадианы также включать в степенной одночлен представления «физического» типа и, например, проверять, что функция SIN берется от величины в радианах или в градусах, а не в метрах. Но тогда формально возникает конфликт размерностей при других вычислениях, например:

```
W=φ/t;    // угловая скорость (угол делится на время)
V=R*W; // линейная скорость (радиус поворота умножается
          // на угловую скорость)
```

Если угол φ имеет «физический» тип «радиан», то получается, что угловая скорость W имеет «физический» тип «радиан в секунду», а линейная скорость после умножения на радиус — «метр на радиан в секунду», что не соответствует требуемой размерности скорости «метр в секунду». Поэтому хотя вспомогательные величины радиан и стерадиан действительно также представлены в степенном одночлене «физического» типа, но ведут себя не совсем так, как семь остальных базовых величин. Они считаются «размерными» величинами, только если остальные величины в одночлене имеют нулевые показатели степени (т.е. отсутствуют). Как только при вычислениях появляется любая другая величина — радиан и стерадиан «сокращаются», т.е. их показатель степени транслятором обнуляется. Тогда в приведенном примере выражение φ/t (и угловая скорость W) получает «физический» тип «единица, деленная на секунду», а выражение R*W правильный «физический» тип скорости «метр в секунду». Но при этом выражение, например, SIN(2*φ) по-прежнему можно проверить на корректный «физический» тип аргумента «радиан».

Поскольку вычисления в «инженерных» задачах проводятся и с дробными степенями, например, Z=SQRT(X**2+Y**2), то показатели степени в универсальном «физическом» типе представлены рациональными (и возможно неправильными) дробями.

При анализе выражений в программе и генерации операций над переменными с «физическими» типами компилятор производит очевидные действия, чтобы получить текущий «физический» тип всего выражения:

— при умножении переменных показатели степеней их «физических» типов складываются; Поскольку степени представлены дробями, то и складываются они как дроби;

— при делении переменных показатели степеней их «физических» типов вычитаются;

— при возведении переменной в степень-константу показатели степеней «физического» типа умножаются на эту константу; При этом возведение в нецелую степень или в степень-переменную для «физического» типа считается ошибкой, за исключением случая представления степени-константы в виде дроби. Например, в выражении X**(1e0/3e0) показатели степени «физического» типа переменной X будут умножены на дробь 1/3, т.е. в данном случае знаменатели степеней будут умножены на три;

— при сложении, вычитании, сравнении и присваивании переменных с «физическими» типами, сами типы не меняются, а просто сравниваются на совпадение; При этом масштабные коэффициенты учитываются в выражениях и присваиваниях и поэтому в этих операциях могут не совпадать.

Однако есть частные случаи присваивания или сравнения (например, векторов), когда присваивание или сравнение реализуется просто побайтным переписыванием или сравнением двух участков памяти. В таких случаях и масштабные коэффициенты «физических» типов у сравниваемых или присваиваемых переменных обязаны также совпадать.

Для задания «физических» типов используются символы квадратных скобок, например: опс V вещ(53) [м/с];

Внутри квадратных скобок могут быть записаны произвольные выражения, состоящие из скобок, знаков умножения, деления, возведения в степень, числовых констант и девяти имен базовых и вспомогательных величин СИ: **м, кг, с, а, к, моль, кд, рад, ср**. Для угловых величин допустимо также имя «ПИ». Например:

```
опс Mu вещ(53) [(1000*m)**3/c**2];
опс Fi вещ(53) [пи/180*рад];
```

Встроенная таблица «физических» типов в трансляторе имеет лишь девять первых заполненных строк перечисленными выше базовыми

величинами, однако с помощью препроцессорного оператора замены %REPLACE/%ЗАМЕНИТЬ (см § 13.6.) в неё можно добавлять произвольное число «нестандартных» типов, например:

```
%заменить
[км]          это [1000*м],
[час]         это [3600*с],
[миля]        это [1852*м],
[узел]        это [миля/час];
описание
скорость            вещ(53) [км/час],
скорость_судна      вещ(53) [узел];
```

При этом после выполнения оператора замены новые имена «км», «час», «миля» и «узел» становятся допустимыми только внутри квадратных скобок описания «физического» типа, в остальном тексте программы эти имена можно использовать для других целей.

Поскольку числовые константы, записываемые непосредственно в физических уравнениях не должны иметь собственной размерности, инициализация значений выполняется так:

```
описание
//---- константы ----
g   вещ(53) значение(9.81e0) [м/с**2],
v0  вещ(53) значение(10e0)   [км/час],
//---- переменные ----
m  вещ(53) [кг],
v   вещ(53) [км/час],
F   вещ(53) [кг*м/с**2];
…
v=v0; F=g*m;
```

Исключение составляет константа 0. Её «физический» тип может быть любым:

```
v=0; F, m=0;
```

Двоичный обмен операторами READ/ВВОД и WRITE/ВЫВОД осуществляется без каких-либо преобразований и поэтому переменные с «физическими» типами записываются и читаются из файлов точно так же, как и любые другие переменные.

А вот в другом случае появляется несколько непривычное свойство вывода переменных с «физическими» типами. Поскольку в общем случае текстового вывода выдаваемый объект — это выражение, то для переменных с «физическими» типами в нем автоматически будут учтены масштабные коэффициенты, а, следовательно, выражение всегда будет выведено в базовых

единицах СИ, например, в метрах, даже когда выводится одна переменная и она задана в километрах или в миллиметрах. Для того чтобы операторы PUT/ПИСАТЬ и GET/ЧИТАТЬ оставались «зеркальными» транслятор автоматически учитывает масштабный коэффициент переменной и при чтении её оператором GET/ЧИТАТЬ, т.е. при чтении также подразумевается, что значения хранятся в файлах в базовых единицах СИ.

Если все же необходим текстовый обмен без учета масштабных коэффициентов, то это можно реализовать, используя определяемые переменные, т.е. описав переменные без «физических» типов по тем же адресам памяти, что и переменные с «физическими» типами, например, в случае:

```
описание
X1 вещ(53) постоянное задать(10e0) [км],
X2 вещ(53) на_месте(X1);
писать с_новой в_виде(X1, X2);
```

будут выданы значения 10000 и 10.

Для удобства вывода введена встроенная переменная (строка) **?TYPE**, автоматически принимающую значение «физического» типа (т.е. размерности) последнего вычисленного выражения, например, при выполнении оператора вывода в данном примере:

```
описание
s  вещ(53) постоянное задать(10) [км],
t  вещ(53) постоянное задать(5)  [час];
…
писать с_новой в_виде(s, ?type,t, ?type, s/t, ?type);
```

будут выданы следующие значения:

```
1.00000000000000E+004 М  1.80000000000000E+004 С  5.55555555555555E-001
М*С**-1
конец программы
```

Параметры процедур, как и обычные переменные, также могут иметь «физический» тип с масштабным коэффициентом. При вычислении и подстановке фактического параметра транслятор выполняет те же действия, что и при обычном присваивании, т.е. сравнивает показатели степеней базовых величин и умножает на масштабный коэффициент. Однако если параметром является не скалярная величина, а сразу агрегат данных, масштабный коэффициент формального и фактического параметров должны строго совпадать и в этом случае на коэффициент ничего не умножается.

Также «физический» тип может иметь возвращаемое подпрограммой значение:

```
описание
//---- модуль радиус-вектора ----
Vmod для_вызова((3) вещ(53)[км]) возвращает(вещ(53)[км]);
```

Поскольку на практике используются и подпрограммы с полиморфными типами, допускается задание в формальных параметрах неопределенного «физического» типа (с помощью знака «?»), при котором проверки на совпадение и учет масштабных коэффициентов отключаются.

```
declare
//---- модуль любого вектора ----
Vmod0 для_вызова((3) вещ(53)[?]) возвращает(вещ(53)[?]);
```

Еще один особый случай представляет встроенная функция квадратного корня SQRT. Она тоже имеет неопределенный «физический» тип формального аргумента, но возвращает всегда такой же тип как у фактического аргумента и в степени ½.

В рассматриваемой версии языка кроме встроенной переменной ?TYPE, есть ещё два отладочных средства. Это возможность вывести всю таблицу переменных программы, включая и их «физические» типы с масштабными коэффициентами и собственно сообщения при компиляции о несоответствии типов в вычисляемых выражениях. В данном случае в небольшом тесте, приведенном ниже, в операторе сравнения «физические» типы справа и слева оказались не равны.

```
test:проц главная;
%заменить
[км]  это [1000*м],
[час] это [3600*с];
описание
s  вещ(53) постоянное задать(10) [км],
t  вещ(53) постоянное задать(5)  [час],
v  вещ(53) [км/час];
если s*t>v*t тогда стоп;
...
```

Вывод таблицы переменных с «физическими» типами и сообщения о несоответствии «физических» типов в операции сравнения:

```
a 00000000 TEST ПРОЦЕДУРА ПАРАМЕТРЫ(0) ОБЩЕЕ НЕИЗМЕНЯЕМОЕ
БЛОК В СТРОКЕ 3, ВЫДЕЛЕНО ПАМЯТИ 8 БАЙТ
c 00000000 S ВЕЩ. ЧИСЛО ДВОИЧНОЕ (53) СО ЗНАЧЕНИЕМ ПОСТОЯННОЕ
[1.0000000E+03*M]
c 00000000 T ВЕЩ. ЧИСЛО ДВОИЧНОЕ (53) СО ЗНАЧЕНИЕМ ПОСТОЯННОЕ
[3.6000000E+03*C]
c 00000000 V ВЕЩ. ЧИСЛО ДВОИЧНОЕ (53) ВРЕМЕННОЕ [2.7777777E-01*M*C**-
1]
```

```
*КОНЕЦ*
  13 c      if s*t>v*t then stop;
РЗМ. НЕ=                  ?
M*C # M
```

Приведем еще ряд примеров изменения программы путем введения для ее переменных «физических» типов.

а) Ввод типов для ряда переменных вместо ранее безразмерных констант, например:

```
// СКОРОСТЬ ВРАЩЕНИЯ ЗЕМЛИ
WEarth  вещ(53) постоянное задать(0.000072921158e0)       [РАД/С],
// ДВА ПИ
Dpi     вещ(53) постоянное задать(6.28318530717958648e0) [РАД],
// Epsilon/Mu
Em      вещ(53) постоянное задать(66072.1866e0)           [КМ**2],
// ГРАВИТАЦИОННЫЙ ПОТЕНЦИАЛ
Mu      вещ(53) постоянное задать(398600.4e0)             [КМ**3/С**2],
// ПОЛУОСЬ ДЛЯ СРЕДНЕЙ ВЫСОТЫ 358 КМ
Am      вещ(53) постоянное задать(6736e0)                 [КМ],
//ЭКВАТОРИАЛЬНЫЙ РАДИУС ЗЕМЛИ
Re      вещ(53) постоянное задать6378.137e0)              [КМ],
```

б) Убирание ряда приведений масштабов переменных, например, было:

```
//---- ПРИВЕДЕНИЕ МАСШТАБОВ К КМ И +-180 ----
цикл i=1 to 3;
     vsg(i)=vsg0(i)/1000e0;         // ПЕРЕВЕЛИ В КМ
     vrg(i)=vrg0(i)/1000e0;         // ПЕРЕВЕЛИ В КМ
     если Lamseans(i) > 180e0 тогда Lamseans(i)-=360e0;
конец i;
```

Стало:

```
//---- ПРИВЕДЕНИЕ МАСШТАБОВ К КМ И +-180 ----
цикл i=1 до 3;
   vsg(i)=vsg0(i);         // АВТОМАТИЧЕСКИЙ ПЕРЕВОД В КМ
   vrg(i)=vrg0(i);         // АВТОМАТИЧЕСКИЙ ПЕРЕВОД В КМ
   если Lamseans(i) > У_180 тогда Lamseans(i)-=У_360;
конец i;
```

Было:

```
BetaBal=60e0*(per2-per1)/(te2s-te1s)/2e0;   // ПЕРИОД БЫЛ В МИНУТАХ
```

Стало:

```
BetaBal=(per2-per1)/(te2s-te1s)/2e0;        // ПЕРИОД БЫЛ В МИНУТАХ
```

в) Некоторые произвольно разбитые на части формулы потребовалось свернуть, чтобы не вводить промежуточных переменных с нестандартной размерностью, например, было:

```
//---- ТЕКУЩИЙ РАДИУС ДЛЯ РАСЧЕТА ВЫСОТЫ ----
Radius  = J3-et1+dz1*dz1+J4*cos(ArgLat+ArgLat);
Radius *= Axe*(1e0+J7*tp);
```

Стало:

```
//---- ТЕКУЩИЙ РАДИУС ДЛЯ РАСЧЕТА ВЫСОТЫ ----
Radius  = Axe*(1e0+J7*tp)*(J3-et1+dz1*dz1+J4*cos(ArgLat+ArgLat));
```

Однако большей частью исходный текст остался без изменений. В результате тестовых расчетов были получены те же результаты, что и у исходной программы, при компиляции сообщений о несоответствии типов не выдавалось, например, автоматически проверено, что самая громоздкая из использованных формул дала корректную размерность длины:

```
//---- ОЦЕНКА ПОЛУОСИ ПО ЗНАЧЕНИЮ ПЕРИОДА ----
цикл AxeOfPer=Am, i=1 to 4;                // НАЧИНАЕМ С ВЫСОТЫ 358 КМ
 AxeOfPer=((Period/Dpi)**2*Mu)**(1e0/3e0)
        /(1e0-2e0/3e0*Em/AxeOfPer/AxeOfPer*(4e0*CosIncl*CosIncl-1e0));
конец;
```

Однако пришлось и устранить в программе один появившийся недостаток:

```
//---- ШИРОТА В ГРАДУСАХ С УЧЕТОМ ЭЛЛИПСОИДНОСТИ ЗЕМЛИ ----
Fi = Fi + Alz*sin(2e0*Fi);
```

В этом операторе Alz — безразмерный коэффициент сжатия земного эллипсоида, равный 1/298.257. Получается, что в формуле угол Fi в радианах складывается с заведомо безразмерным синусом, т.е. несоответствие размерности. Так сказалось формальное представление углов условно размерной величиной «радиан», несмотря на то, что реальный радиан величина безразмерная. Для преодоления этого несоответствия в формуле пришлось дополнительно умножить синус на переменную с «физическим» типом «радиан» и единичным значением.

Заметим, что с помощью оператора замены легко добавить и производные величины СИ:

```
%заменить
[Гц] это [1/с],                 // герц      Частота
[Н]  это [м*кг/(с*с)],          // ньютон    Сила, вес
[Па] это [кг/м/(с*с)],          // паскаль   Давление, мех напряжение
[Дж] это [м*м*кг/(с*с)],        // джоуль Энергия, работа, кол.теплоты
[Вт] это [м*м*кг/(с*с*с)],      // ватт      Мощность
[Кл] это [с*А],                 // кулон     Количество электричества
[В]  это [м*м*кг/(с*с*с)/А],    // вольт Напряжение, потенциал, э.д.с.
[Ф]  это [(с*с*с*с)*А*А/(м*м)/кг],// фарада  Электрическая ёмкость
[Ом] это [м*м*кг/(с*с*с)/(А*А)],// ом   Электрическое сопротивление
[См] это [(с*с*с)*А*А/(м*м)/кг],// сименс   Электрическая проводимость
[Вб] это [м*м*кг/(с*с)/А],      // вебер     Поток магнитной индукции
[Тл] это [кг/(с*с)/А],          // тесла     Магнитная индукция
[Гн] это [м*м*кг/(с*с)/(А*А)],  // генри     Индуктивность
[лм] это [кд*ср],               // люмен     Световой поток
[лк] это [кд*ср/(м*м)],         // люкс      Освещённость
[Бк] это [1/с],                 // беккерель Активность радионуклида
[Гр] это [м*м/(с*с)],           // грей     Поглощ. доза ионизир. изл.
[Зв] это [м*м/(с*с)];           // зиверт Эквивалентная доза излучения
```

Проверь себя

1. Что такое «физические» типы, и на каком этапе создания программы они используются?
2. Сколько независимых «физических» типов с точки зрения системы единиц СИ имеется? Сколько дополнительных типов можно ввести в программу?
3. Каким оператором можно ввести «физический» тип «дюйм»?
4. Зачем нужны масштабные коэффициенты в «физических» типах?
5. Можно ли используя «физические» типы исключить применение встроенных функций SIND, COSD, TAND?
6. В чем разница записи в файл и чтения из файла переменных, имеющих «физический» тип и всех остальных?
7. Для чего нужна и используется встроенная переменная ?TYPE и какие значения она принимает?
8. Переменная *x* имеет «физический» тип [m]. Какой «физический» тип должна иметь переменная *y*, чтобы оператор y=sqrt(x); был корректным?

13.4. Использование операторов ввода-вывода для передачи данных в памяти

Операторы потокоориентированного ввода (GET/ЧИТАТЬ) и вывода (PUT/ПИСАТЬ) могут быть использованы для установки значений переменных на основе значений других переменных, без использования внешней памяти (т.е. файлов). В этом случае операторы должны иметь вид

k* STRING(*c*) *d или ***k* ИЗ_СТРОКИ(*c*) *d*** или ***k* В_СТРОКУ(*c*) *d***

Здесь k — ключевое слово GET/ЧИТАТЬ или PUT/ПИСАТЬ; d — одна из конструкций LIST/В_ВИДЕ s_l, либо EDIT/В_ФОРМЕ s_e, либо DATA/С_ИМЕНАМИ s_d.

где s_l , s_e и s_d — спецификации, имеющие тот же вид; что и при использовании операторов для ввода или вывода; c — имя переменной символьно-строчного типа.

Оператор GET/ЧИТАТЬ с конструкцией STRING/ИЗ_СТРОКИ(*c*) выполняется таким образом, как если бы значение c было бы очередным фрагментом входного потока. Например, если значение переменной T равно строке символов '1 ␣ 253', то после выполнения оператора

```
get string (T) edit(x, y, z)(f(1), x (1), f(2)); или
читать из_строки (T) в_форме(x, y, z)(ч(1), п(1), ч(2));
```

значения числовых переменных X, Y и Z станут, соответственно, равны 1, 25 и 3.

При выполнении оператора GET/ЧИТАТЬ с конструкцией STRING/ИЗ_СТРОКИ никогда не возникает состояние ENDFILE/КОНЕЦ_ФАЙЛА. Вместо этого в соответствующих ситуациях возникает состояние ошибки с соответствующим значением встроенной функции ONCODE/КОГДА_КОД.

Оператор PUT/ПИСАТЬ с конструкцией STRING/В_СТРОКУ(*c*) задает присваивание переменной *c* такой строки символов, которая бы была сформирована в выходном потоке при обычном выводе. Например, в результате выполнения операторов:

```
опс t текст (100) рд,
(x, y, z) точное;
x = -5; y = 6; z = 77;
писать в_строку(t) в_форме (x, y, z) (ч (3));
```

переменная T примет значение, равное строке символов « ␣ -5 ␣␣ 6 ␣ 77 »

В списках форматов в операторах GET/ПИСАТЬ и PUT/ЧИТАТЬ с конструкцией STRING, ИЗ_СТРОКИ или В_СТРОКУ нельзя использовать форматы PAGE, LINE, SKIP и COLUMN или ПЕРЕВОД_СТРАНИЦЫ, ПЕРЕВОД_СТРОКИ, С_НОВОЙ и СТОЛБЕЦ.

Отметим, что удобно использовать эту конструкцию для разбора параметров главной процедуры, запущенной из командной строки Windows. Напомним, что главная процедура может иметь единственный параметр в виде символьно-строчной переменной. Пусть в командной строке вызова нужно задать три параметра в виде чисел через пробелы. Это можно сделать следующими операторами.

```
P:проц(S) главная;
опс S текст(*) рд, (X,Y,Z) вещ;
читать из_строки(S) в_виде(X, Y, Z);
```

Проверь себя

1. Какой вид должны иметь операторы потокоориентированного ввода или вывода, если они используются для передачи данных в оперативной памяти?

2. Указать, какие значения примут переменные X, Y и Z после выполнения фрагмента программы

```
опс T текст(10), (X, Y, Z) точное;
T = '55.67'; читать в_строку(T) в_форме(X, Y, Z)(ч(2,1));
```

3. Какое состояние возникнет при выполнении фрагмента программы

```
опс t текст(10), a(11) вещ; t='9'(10);
читать из_строки(t) в_форме(a)(f(1));
```

4. Указать, какое значение примет переменная T после выполнения фрагмента программы

```
опс t текст(100)рд, (x, y, z) точное(5, 2);
x=-2; y = 0.99; z = 3;
писать в_строку(t) в_форме(x, y, z)(ч(4, 1));
```

13.5. Внутреннее представление данных

Данные любого типа представляются в памяти в виде последовательности из определенного числа битов. Например, символ «R» представляется в виде последовательности битов 01010010 или '52'Б4.

В языке PL/1 предусмотрены встроенная функция и псевдопеременная UNSPEC/МАШ_КОД, позволяющие манипулировать внутренним представлением данных. Значение функции UNSPEC/МАШ_КОД(*e*), где *e* — имя переменной со значением любого типа, равно строке битов, идентичной внутреннему представлению значения *e*. Например (см. выше), если выполнить операторы

```
опс S текст(1); S='R'
писать с именами(маш_код(S));
```

то получим ответ: `UNSPEC(S)= '01010010'b`

Псевдопеременной UNSPEC/МАШ_КОД (*u*), где *u* — переменная любого типа, присваивается значение, являющееся произвольной строкой битов *t*.

В результате подобного присвоения внутреннее представление значения переменной *u* становится равным строке *t*, возможно усеченной справа или дополненной справа нулевыми битами до требуемой длины. Например, если выполнить оператор дополнительно к приведенным выше маш_код(s)= '45'Б4;

То переменная S получит значение «E», поскольку его внутреннее представление 01000101.

В зависимости от заданной точности, данные FIXED BINARY/ТОЧНОЕ ДВОИЧНОЕ могут занимать один, два, четыре или восемь байт памяти. Для процессоров архитектуры x86 действует правило: младший байт слова находится по младшему адресу, таким образом константа '1234'b4 будет помещена в двух соседних байтах как 34 12. Старший бит в данных обозначает знак, если число отрицательное - оно пишется в дополнительном коде. Примеры чисел:

FIXED BINARY(7) или ТОЧНОЕ ДВОИЧНОЕ(7)

Число	представление
0	00
1	01
-1	FF
127	7F

FIXED BINARY(15) или ТОЧНОЕ ДВОИЧНОЕ(15)

Число	представление
0	00 00
1	01 00
-1	FF FF
127	7F 00

FIXED BINARY (31) или ТОЧНОЕ ДВОИЧНОЕ(31)

Число	представление
0	00 00 00 00
1	01 00 00 00
-1	FF FF FF FF
127	7F 00 00 00

FIXED BINARY(63) или ТОЧНОЕ ДВОИЧНОЕ(63)

Число	представление
0	00 00 00 00 00 00 00 00
1	01 00 00 00 00 00 00 00
-1	FF FF FF FF FF FF FF FF
127	7F 00 00 00 00 00 00 00

Представление FLOAT BINARY и FLOAT DECIMAL или ВЕЩЕСТВЕННОЕ ДВОИЧНОЕ и ВЕЩЕСТВЕННОЕ ДЕСЯТИЧНОЕ

В формате IEEE-754 и двоичная и десятичная формы представляются одинаково.

Для чисел с обычной точностью биты используются следующим образом:

	знак	порядок	мантисса
Номера бит	31	30-23	22-0

В формате IEEE-754 мантисса всегда нормализована, т.е. умножена так, чтобы ее первый разряд всегда был 1, раз это так, этот разряд можно не хранить. Такой бит мантиссы называется неявным. Двоичная точка находится непосредственно справа от неявного бита. Порядок в формате IEEE-754 сдвинут на 127 (или 7F), так, что 80 означает +1, 7E означает -1 и т.д.

Разберем число в формате IEEE-754: 00 00 C0 3F. Как битовая строка, это число выглядит так:

3 F C 0 0 0 0 0
0011 1111 1100 0000 0000 0000 0000 0000
Старший бит показывает, что число положительное, а порядок без смещения 7F дает 0:
00111 1111 1000 0000 0000 0000 000
знак порядок мантисса
С учетом неявного бита и точки, мантисса выглядит так:
1. 100 0000 0000 0000 0000 0000
Переводим в десятичный вид:
2**0 + 2**-1 + 0 =1.5
Таким образом, 00 00 C0 3F в формате IEEE-754 это 1.5.

Для чисел с двойной точностью биты используются следующим образом:

	знак	порядок	Мантисса
Номера бит	63	62-52	51-0

Порядок здесь сдвинут на 1023 (или 3FF), так, что 400 означает +1, 3FE означает -1 и т.д. Разберем число двойной точности: 00 00 00 00 00 C0 43 C0.
C 0 4 3 C 0 0 0
1100 0000 0100 0011 1100 0000 0000 0000 ...
знак порядок мантисса
С учетом неявного бита и точки, мантисса выглядит так:
1. 0011 1100 0000 0000 0000 0000 ...
Порядок:
40 4
0100 0000 0100 отнимаем 3FF, получаем 5
Учитываем порядок:
10011 1.100 0000 0000 0000 0000 ...
Переводим в десятичный вид:
2**5 + 2**2 + 2**1 + 2**0 + 2**-1 = 39.5 и учитываем знак минус
Таким образом, 00 00 00 00 00 C0 43 C0 в формате IEEE-754 это -39.5.

Представление FIXED DECIMAL или ТОЧНОЕ ДЕСЯТИЧНОЕ

Данные представляются в десятичном дополнительном упакованном коде BCD (Binary Coded Decimal). Каждая цифра BCD записывается в половину байта. PL/1 помещает младшую значащую пару цифр BCD по старшему адресу. Первая позиция BCD резервируется для знака. Знак для положительных чисел — ноль, для отрицательных — девять. Число байт определяется заданной точностью и равно FLOOR((*p*+2)/2), где *p* допустима от 1 до 15. Для примера, рассмотрим число 12345 с точностью 5, для которого PL/1 выделит FLOOR((5+2)/2)=3 байта памяти. Число будет записано как: 45 23 01. Отрицательное число записывается в дополнительном до 9 коде, дополнительный код здесь получается вычитанием из 9 и прибавлением единицы. Например, -2 имеет код (9-2)+1=8 и с точностью 2 будет записано

как 98, а с точностью 5 будет выглядеть: 98 99 99.

Представление CHARACTER или ТЕКСТ

Данные представляются в двух видах:

— для строки фиксированной длины, байты символов помещаются последовательно от младшего адреса к старшему;

— для строки переменной длины (VARYING/РАЗНОЙ_ДЛИНЫ), сначала пишется байт длины (от 0 до 254), а затем байты символов, как в предыдущем случае. Таким образом, строка переменной длины занимает число байт, равное максимальной длине плюс еще один байт длины.

Например, строка 'Walla Walla Wash' в первом случае будет представлена как:

W a l l a W a l l a W a s h
57 61 6C 6C 61 20 57 61 6C 6C 61 20 57 61 73 68

а во втором случае:

W a l l a W a l l a W a s h
10 57 61 6C 6C 61 20 57 61 6C 6C 61 20 57 61 73 68

Представление BIT/БИТ

Битовые строки, в зависимости от заданной длины, представляются или одним (1-8 битов) байтом или двумя байтами (9-16 битов) или четырьмя (17-32 бита) или восьмью (33-64 бита) и всегда начинаются на границе байта, даже, если их длина не кратна 8 битам. Напомним, что действует правило: младший байт слова находится по младшему адресу. Если число бит не равно 8 или 16 или 32, лишние биты справа игнорируются. Примеры представления строк:

Запись в PL/1	Описание	Представление в памяти
'1'б	**БИТ (8)**	00000001
'1'б	**БИТ (16)**	0000001 00000000
'A0'б4	**БИТ (8)**	10100000
'A0'б4	**БИТ (16)**	10100000 00000000
'1234'б4	**БИТ (16)**	00110100 00010010
'12345678'б4	**БИТ (32)**	01111000 01010110 00110100 00010010

Представление POINTER/УКАЗАТЕЛЬ

Данные представляются в восьми байтах - там находится адрес памяти переменной.

Представление ENTRY/ДЛЯ_ВЫЗОВА и LABEL/МЕТКА

Данные представляются в восьми байтах для LABEL/МЕТКА (адрес метки) и для ENTRY/ДЛЯ_ВЫЗОВА (адрес входа в процедуру).

Представление FILE/ФАЙЛ

Каждый идентификатор файла в PL/1 ассоциируется со статически выделяемой для него структурой *блока параметров файла* (FPB) и динамически выделяемой (во время открытия) и освобождаемой (во время закрытия) структурой *управляющего блока файла* (FCB), в конце которой размещается и буфер обмена для данного файла, если файл отображается на память — буфера нет.

Файл фактически представляет собой указатель на структуру FPB, причем после открытия файла, первые восемь байт FPB содержат указатель на FCB. Можно описать переменные типа POINTER/УКАЗАТЕЛЬ и оператором инициации (INIT/ЗАДАТЬ) присвоить им значения файлов. Напомним, что напрямую пересылать значения файлов можно только в переменные типа FILE VARIABLE или ФАЙЛ КОСВЕННОЕ.

Структура блока параметров файла (FPB)

Смещ.	Длина	Содержимое
+00	8	указатель на FCB, результат выделения памяти внутри OPEN/ОТКРЫТЬ. Показывает на выделенные 35H+<длина буфера> байт, после CLOSE эта память опять освобождается
+08	2	позиция в строке для STREAM/ТЕКСТОВЫЙ, первоначально 1
+0A	2	номер строки для STREAM/ТЕКСТОВЫЙ, первоначально 1
+0C	2	номер страницы для STREAM/ТЕКСТОВЫЙ, первоначально 1
+0E	1	считанный, но еще не обработанный символ
+0F	8	значение стека (RSP) после возврата из ON/КОГДА-оператора
+17	8	адрес возврата после ON/КОГДА-оператора
+1F	2	размер строки, берется из LINESIZE/С_ДЛИНОЙ_СТРОКИ
+21	2	размер страницы, берется из PAGESIZE/С_ДЛИНОЙ_СТРАНИЦЫ
+23	4	размер записи для KEYED/ИНДЕКСНЫЙ, берется из ENV/ДЛЯ_ОС(F)
+27	4	размер буфера, берется из ENV/ДЛЯ_ОС(B), если -1 - то буфера нет и файл отображается на память. Если буфер задан, он не может быть короче размера записи (если запись есть)
+2B	2	атрибуты файла из OPEN/ОТКРЫТЬ (см. следующую таблицу)
+2D	1	длина внутреннего (в PL/1) имени файла
+2E	1-31	внутреннее (в PL/1) имя файла

Номер бита в поле 2B	Признак атрибута
XXXX XXXX XXXX XXX**1**	ENV/ДЛЯ_ОС (опция A)
XXXX XXXX XXXX XX**1**X	PRINT/ДЛЯ_ПЕЧАТИ
XXXX XXXX XXXX X**1**XX	KEYED/ИНДЕКСНЫЙ
XXXX XXXX XXXX **1**XXX	DIRECT/ПРЯМОЙ
XXXX XXXX XXX**1** XXXX	SEQUENTIAL/ПООЧЕРЕДНЫЙ
XXXX XXXX XX**1**X XXXX	UPDATE/ДЛЯ_ИЗМЕНЕНИЙ
XXXX XXXX X**1**XX XXXX	OUTPUT/ДЛЯ_ВЫВОДА
XXXX XXXX **1**XXX XXXX	INPUT/ДЛЯ_ВВОДА
XXXX XXX**1** XXXX XXXX	RECORD/НЕ_ТЕКСТОВЫЙ
XXXX XX**1**X XXXX XXXX	STREAM/ТЕКСТОВЫЙ
XXXX X**1**XX XXXX XXXX	FILE/ФАЙЛ
XXXX **1**XXX XXXX XXXX	TITLE/С_ИМЕНЕМ
XXX**1** XXXX XXXX XXXX	LINESIZE/С_ДЛИНОЙ_СТРОКИ
XX**1**X XXXX XXXX XXXX	PAGESIZE/С_ДЛИНОЙ_СТРАНИЦЫ
X**1**XX XXXX XXXX XXXX	ENVIROMENT/ДЛЯ_ОС

Можно описать приведенную ниже структуру для непосредственного

доступа ко всем внутренним данным переменной типа файл.
описание

```
 1 STRUC_FPB    основа,
  2 FCB         указ,       /* указатель на FCB */
  2 COL         точное,     /* текущий номер позиции в строке */
  2 LINENO      точное,     /* текущий номер строки */
  2 PAGENO      точное,     /* текущий номер страницы */
  2 SYMBOL      текст,      /* считанный, еще не обработанный символ */
  2 SP_ON       бит(64),    /* стек после возврата из ON-оператора */
  2 RET_ON      бит(64),    /* адрес возврата после ON-оператора */
  2 LINESIZE    точное,     /* размер строки */
  2 PAGESIZE    точное,     /* размер страницы */
  2 F           точное(31), /* размер записи */
  2 B           точное(31), /* размер буфера */
  2 ATTRIBUTS   бит(16),    /* атрибуты */
  2 INT_NAME    текст(31) рд; /* имя файла в PL/1 */
```

Структура управляющего блока файла (FCB)

Смещ.	Длина	Содержимое
+0	1	тип файла 0 - ввод, 1 - вывод, 2 – изменения
+1	1	драйвер 0 - текущий, 1 - A:, 2 - B: и т.д.
+2	8	внешнее имя файла в стиле ДОС (дополняется до 8 пробелами)
+A	3	расширение файла в стиле ДОС (дополняется до 3 пробелами)
+D	4	описатель открытого файла (HANDLE)
+11	4	размер записи для работы по ключу (KEY/ИНДЕКС) или для KEYTO/ДЛЯ_ИНДЕКСА
+15	4	счетчик в буфере, когда достигает размера буфера, производится обмен буфера с диском, после чего этот счетчик сбрасывается в ноль. Если счетчик ноль и текущий размер буфера ноль - возникает ситуация ENDFILE/КОНЕЦ_ФАЙЛА
+19	4	текущий размер буфера, после очередного обмена сюда пишется действительное число считанных или записанных байт
+1D	4	размер файла в байтах при открытии
+21	4	текущий номер записи (значение KEY/ИНДЕКС или KEYTO/ДЛЯ_ИНДЕКСА)
+25	4	адрес нулевой записи в файле, каждый раз прибавляется к адресу записи (т.е. к размеру записи, умноженному на номер записи). При открытии файла устанавливается в ноль.
+29	8	смещение буфера, при открытии устанавливается на FCB+35H
+31	4	число записей в файле, устанавливается один раз при открытии, корректируется при записи оператором WRITE KEYFROM/ВЫВОД ИЗ_ИНДЕКСА
+35	...	буфер при открытии, если был задан -1, то эта память для него не выделяется

Можно описать приведенную ниже структуру для непосредственного доступа ко всем внутренним данным этой, динамически создаваемой, структуры.

```
описание
 1 STUC_FCB     основа,
  2 TYPE        точное(7),   /* 0 - ввод, 1 - вывод, 2 - изменение */
  2 DRV         точное(7),   /* драйвер 0 - текущий, 1 - A: ... */
  2 NAME_1      текст(8),    /* внешнее имя файла */
  2 NAME_2      текст(3),    /* расширение файла */
  2 HANDLE      точное(31), /* номер открытого файла */
  2 REC_SIZE    точное(31), /* размер записи */
  2 COUNT       точное(31), /* текущая позиция в буфере */
  2 BUF_SIZE    точное(31), /* текущий размер буфера */
  2 SIZE        точное(31), /* размер файла при открытии */
  2 N_REC       точное(31), /* текущий номер записи */
  2 N0_OFF      бит(32),     /* смещение нулевой записи */
  2 BUF_OFF     бит(64),     /* смещение буфера */
  2 MAX_REC     точное(31); /* число записей в файле */
```

Пример доступа к внутренним структурам данных типа файл.

```
описание
P1          указ постоянное задать(FILE_1),
P2          указ постоянное задать(FILE_2),
FPB         указ,
FILE_1      файл,
FILE_2      файл;
...
открыть файл(FILE_1) для_печати с_именем('C:\TEST\README.TXT');
FPB=P1;
писать с_новой в_виде(FPB->LINENO);
писать с_новой в_виде(FPB->PAGENO);
писать с_новой в_виде(FPB->FCB->REC_SIZE);
писать с_новой в_виде(FPB->FCB->COUNT);
...
открыть файл(FILE_2) для_ввода с_именем('$1.$1');
FPB=P2;
писать с_новой в_виде(FPB->INT_NAME);
писать с_новой в_виде(FPB->FCB->SIZE);
...
```

Размещение агрегатов данных

PL/1 пишет агрегаты данных последовательными участками байт. Массив из БИТ (1) не будет выравниваться, но каждый бит будет писаться с начала нового байта. Элементы массива располагаются последовательно, так, что правый индекс меняется быстрее всего. Например, массив A(2,2,2) расположится в памяти:

1,1,1	1,1,2	1,2,1	1,2,2	2,1,1	2,1,2	2,2,1	2,2,2
1	2	3	4	5	6	7	8

Внутреннее представление формата EDIT/В_ФОРМЕ

Формат начинается байтом 80, после которого следует коэффициент повтора, затем собственно данные формата и байт конца 81. Данные формата состоят из байта типа и затем одного или двух байтов значений. В случае шаблона за типом следует сама строка шаблона.

Тип	элемент	параметры
0	F или Ч	Два байта – общая длина и длина дробной части
1	E	Два байта – общая длина и длина дробной части
2	P или Ш	Байт длины строки и затем сама строка шаблона
3	A или Т	Байт длины
4	B или Б	Байт длины
5	B1 или Б1	Байт длины
6	B2 или Б2	Байт длины
7	B3 или Б3	Байт длины
8	B4 или Б4	Байт длины
9	TAB	Байт количества табуляций
A	LINE/ЗАДАТЬ_СТРОКУ	Байт номера строки
B	X или П	Байт числа пробелов
C	SKIP или С_НОВОЙ	Байт числа пропусков
D	COL или СТОЛБЕЦ	Байт номера позиции
E	R	-
F	PAGE/ ЗАДАТЬ_СТРАНИЦУ	Байт числа пропусков страниц

Байт FF означает значение, принятое по умолчанию.

Проверь себя

1. Укажите назначение встроенной функции и псевдопеременной UNSPEC/МАШКОД.
2. Сколько байт памяти занимает переменная типа ТОЧНОЕ ДЕСЯТИЧНОЕ(15, 0)?
3. Сколько байт памяти занимает переменная типа ВЕЩЕСТВЕННОЕ(53) КОМПЛЕКСНОЕ?
4. Сколько выделяется байт памяти для переменной типа ТЕКСТ(254) РАЗНОЙ_ДЛИНЫ?
5. Всегда ли битово-строчный переменные начинаются на границе байта?
6. Сколько байт памяти занимает переменная типа входа в процедуру?
7. Как поместить в переменную типа УКАЗАТЕЛЬ значение типа ФАЙЛ?

13.6. Подготовка исходных текстов программ

Для удобства формирования текстов программ в большинстве языков программирования (включая PL/1) предусмотрена *препроцессорная обработка* исходного текста с помощью специальной программы-препроцессора. Часто эта программа начинает работать до начала работы собственно компилятора («процессора»), выдавая ему уже как-то переработанный текст, почему она и получила такое название. В рассматриваемой версии языка отдельного препроцессора нет, однако имеется два средства, позволяющих программисту улучшить работу с исходным текстом программы. Чтобы отличать эти средства от обычных операторов программы, они всегда начинаются с символа % (процент). Напоминаем, что в тексте программы можно задавать и некоторые параметры компиляции (см. § 11.5.), которые тоже начинаются с символа процента и которые тоже в некотором роде можно отнести к препроцессорной обработке. Сами препроцессорные средства сводятся к оператору **%INCLUDE** или **%ВСТАВИТЬ_ИЗ** и к оператору **%REPLACE** или **%ЗАМЕНИТЬ**.

Оператор %INCLUDE *c* или %ВСТАВИТЬ_ИЗ *c*, где *c* — символьно-строчная константа в апострофах, предназначен для подстановки в ту точку исходного текста, где стоит этот оператор фрагмента текста из другого файла. Имя этого файла (возможно с указанием папки) указывается в константе c, после константы ставится точка с запятой, например:

%INCLUDE 'c:\test\my_test.txt'; или %ВСТАВИТЬ_ИЗ 'c:\test\my_test.txt';

Этот оператор может быть закомментирован однострочным или многострочным комментарием. Нельзя указывать %INCLUDE/ %ВСТАВИТЬ_ИЗ внутри текста, который сам был вызван этим оператором, т.е. эти операторы не могут быть вложенными друг в друга. Имя и расширение файла с фрагментом текста может быть любым, если путь к папке не указан — читается файл из текущей папки. В рассматриваемой версии языка в имени файла могут быть указаны символы «?» (вопросительный знак), которые заменяются на символ, указанный в реестре Windows (см. раздел 11.5). Кроме этого, если указана буква диска «Z», то папка не указывается, а файл читается не с диска, а прямо из файла PLINK64.EXE (из раздела ресурсов), например:

%ЧИТАТЬ_ИЗ 'Z:SERVICE.DCL';

Здесь из файла PLINK64.EXE читается стандартное описание служебной библиотеки. У таких описаний расширение всегда .DCL.

Этот же оператор может быть использован для указания, какие библиотеки объектных модулей должны быть использованы редактором связей по умолчанию, без их явного указания на этапе редактирования связей.

В этом случае указывается файл (без имени папки) с расширением .L86 и обычно с буквой «Z», показывающей, что и эту библиотеку редактор связей должен читать из ресурсов файла PLINK64.EXE. Например,

```
%ЧИТАТЬ_ИЗ 'Z:SERVICE.L86';
```

Поскольку у стандартных библиотек имя файла с описанием и имя самой библиотеки всегда совпадают, вместо пары операторов подстановки для имени .DCL и имени .L86, можно писать единый оператор, указывая два символа процента, например, оператор:

```
%%ЧИТАТЬ_ИЗ 'Z:SERVICE.DCL';
```

указывает, что при компиляции в эту точку нужно подставить текст из файла SERVICE.DCL, а на этапе редактирования связей использовать библиотеку SERVICE.L86.

Удобно использовать оператор подстановки, когда в разных программах используется один и тот же фрагмент, например одна программа пишет в файл множество переменных, как результат своей работы, а другая должна их прочитать. В этом случае в одной программе можно написать оператор

```
писать в_файл(f) в_виде(%вставить_из 'parametrs.txt';);
```

а в другой программе

```
читать из_файла(f) в_виде(%вставить_из 'parametrs.txt';);
```

и данные будут читаться строго в том порядке, как они писались, а при изменении файла parametrs.txt и перекомпиляции список ввода и вывода одинаково изменится. Обратите внимание, что в примере стоят две точки с запятой — одна относится к подстановке, а другая — к операторам ввода и вывода.

Оператор %REPLACE или %ЗАМЕНИТЬ предназначен для замены идентификатора или константы в тексте программы на другой идентификатор. Например,

```
%REPLACE ИСТИНА BY '1'B; или %ЗАМЕНИТЬ ИСТИНА ЭТО '1'Б;
```

после этого в тексте программы все идентификаторы ИСТИНА при компиляции будут восприниматься как битово-строчная константа с единичным значением. На оператор замены накладываются типичные ограничения препроцессорной обработки. Во-первых, к моменту первого использования имени в тексте программы, оно уже должно быть указано в операторе замены, т.е. здесь правило, что описания могут быть где угодно, не действует. Во-вторых, сам оператор замены должен находиться в самом внешнем блоке. Поэтому обычно операторы замены идут сразу после заголовка главной или внешней процедуры.

Несколько операторов замены могут быть объединены в один через запятую. С помощью операторов замены текст программы может быть

упрощен, кроме этого, становится возможным изменение констант только в одном месте, а не поиском по всему тексту.

Поскольку имена можно заменять на другие имена — становится возможным вводить синонимы ключевых слов. В рассматриваемой версии языка имеется встроенный оператор замены, позволяющий использовать как русские, так и английские ключевые слова, в том числе одновременно. Содержимое этого оператора можно посмотреть при компиляции с параметром «S». Также содержимое этого оператора приведено в приложении, как в алфавитном порядке английских ключевых слов, так и в алфавитном порядке русских ключевых слов. Некоторые ключевые слова заданы сразу в нескольких формах для лучшей читаемости текста программы. Например, ФАЙЛ, ИЗ_ФАЙЛА, В_ФАЙЛ — эквивалентны.

Еще одной формой обработки текста программ является использование в рассматриваемой версии языка *предопределенных констант* — обычно текстовых констант, значения которым присваивает компилятор или редактор связей. Имена таких констант всегда начинаются с символа «?», чтобы они не совпадали с именами переменных программиста.

Имеется предопределенная константа **?DATE** (не путать со встроенной функцией DATE). На этапе редактирования связей в нее подставляется текущая дата и, таким образом, можно выводить дату текущей версии программы оператором вида:

```
писать с_новой в_виде('ВЕРСИЯ от',?DATE);
```

Предопределенная константа **?MD** — заменяется в тексте программы на имя компилируемого модуля. Например:

```
писать с_новой в_виде('Модуль',?MD);
```

Предопределенная константа **?FL** — заменяется в тексте программы на имя компилируемого файла. Пример использования:

```
писать с_новой в_виде('Файл',?FL);
```

Предопределенная константа **?VR** или **?VER** — заменяется в тексте программы на номер версии компилятора в виде текста. Пример использования: `писать с_новой в_виде('Версия',?ver);`

Предопределенная константа **?LN** или **?LINE** — заменяется в тексте программы на текущий номер строки компилируемого модуля в виде текста. Пример использования: `писать с_новой в_виде(' Строка ',?LN);`

Предопределенная константа **?PROC** — заменяется в тексте программы на текущее имя компилируемой процедуры. Пример использования:

```
писать с_новой в_виде('Процедура',?PROC);
```

Проверь себя

1. Какие средства подготовки исходного текста для языка PL/1 может использовать программист? Каковы их преимущества?
2. Каким символом операторы подготовки текста выделяются среди остальных операторов программы?
3. Можно ли вкладывать операторы подстановки фрагментов текста друг в друга?
4. Можно ли размещать оператор замены в конце исходного текста программы?
5. Допустим ли оператор замены %ЗАМЕНИТЬ ПЯТЬ ЭТО 3+2;
6. Что такое предопределенные константы? В каких операторах целесообразно их использовать?

13.7. Особенности работы с файлами

В большинстве случаев работа с файлами определяется возможностями среды Windows. Программист может прямо вызывать API-функции работы с файлами, однако использование файлов через системную библиотеку языка и в терминах PL/1 во многих случаях удобнее и проще.

В рассматриваемой версии языка существует несколько особенностей работы с файлами. Так, в начале работы программы системная процедура пытается выделить в командной строке запуска главной процедуры два имени файла (и два их расширения), разделенные пробелами. Если это удалось — имена и расширения помещаются во внутренние переменные. Даже, если у главной процедуры нет параметра, в операторах открытия файлов можно использовать имена файлов из командной строки как $1 и $2, например:

```
открыть файл(f) для_ввода по_имени('$1.$1');
```

задает открытие файла с именем и расширение первого файла из командной строки запуска;

```
открыть файл(f) для_ввода по_имени('$2.$2');
```

задает открытие файла с именем и расширение второго файла из командной строки запуска;

```
открыть файл(f) для_ввода по_имени('$1.$2');
```

задает открытие файла с именем первого файла и расширением второго файла из командной строки запуска;

```
открыть файл(f) для_ввода по_имени('$1.txt');
```

задает открытие файла с именем первого файла из командной строки запуска и расширением .TXT.

Файлы-стандартные устройства начинаются с символа $, так, имя файла «$con» означает экран при выводе и клавиатуру при вводе. Стандартные файлы ввода и вывода PL/1 неявно открываются в начале работы программы операторами для ввода:

```
open file (SYSIN) stream input environment(B(128)) title('$con') linesize(80);
открыть файл (SYSIN) текстовый для_ввода для_ос(B(128)) с_именем('$con')
с_длиной_строки(80);
```

для вывода:

```
open file (SYSPRINT) stream output print environment(B(128)) title('$con')
linesize(80) pagesize(0);
открыть файл (SYSPRINT) текстовый для_вывода для_печати для_ос(B(128))
с_именем('$con') с_длиной_строки(80) с_длиной_страницы(0);
```

Можно явно задать операторы открытия с другими параметрами и тогда ввод и вывод будет другим, например, не на экран, а в файл и т.п.

Удобные возможности предоставляет работа с файлом «отображенным на память» в терминах Windows, когда подключается стандартный механизм отображения виртуальной памяти с помощью файла-образа на диске.

Это позволяет обращаться с файлом как с массивом или массивом структур в памяти. Рассмотрим следующий пример. Старые программы на Паскале, созданные много лет назад, дают ошибку определения скорости компьютера из-за существенного роста быстродействия за прошедшие годы. Требуется исправить имеющиеся EXE-файлы поиском по контексту ошибочного действия. Программа исправления выглядит так:

```
CORRP:ПРОЦ ГЛАВНАЯ;
ОПС F  ФАЙЛ;
ОПС ВВ БИТ(8) ОСНОВА(Р) DIM(0:1000_000); // ОБРАЗ EXE-ФАЙЛА КАК БАЙТЫ
ОПС Р  УКАЗ;                             // УКАЗАТЕЛЬ НА ОБРАЗ
ОПС I  ТОЧНОЕ(31);
//---- ЕСЛИ ЗАДАНО НЕВЕРНОЕ ИМЯ ФАЙЛА ----
КОГДА КОНЕЦ_ФАЙЛА(F) ПИСАТЬ С_НОВОЙ В_ВИДЕ('FILE NOT FOUND',СТОП);
//---- ОТКРЫВАЕМ ФАЙЛ "ОТОБРАЖЕНИЕМ НА ПАМЯТЬ" ----
ОТКРЫТЬ ФАЙЛ(F) ДЛЯ_ИЗМЕНЕНИЙ НЕ_ТЕКСТОВЫЙ С_ИМЕНЕМ('$1.$1') ДЛЯ_ОС(В(-
1)); // ИМЯ ИЗ КОМАНДНОЙ СТРОКИ
//---- ПОЛУЧАЕМ УКАЗАТЕЛЬ НА ОБРАЗ ФАЙЛА В ПАМЯТИ ----
ВВОД ИЗ_ФАЙЛА(F) В_УКАЗ(Р);
//---- ИЩЕМ СТАНДАРТНЫЙ КОНТЕКСТ ПО ВСЕМУ ФАЙЛУ .EXE ----
┌─ЦИКЛ I=32 ДО ДЛИНА(F)-100;
│   ЕСЛИ ВВ(I+0)^='F7'Б4 ТОГДА ОПЯТЬ;
│   ЕСЛИ ВВ(I+1)^='D0'Б4 ТОГДА ОПЯТЬ;
│   ЕСЛИ ВВ(I+2)^='F7'Б4 ТОГДА ОПЯТЬ;
│   ЕСЛИ ВВ(I+3)^='D2'Б4 ТОГДА ОПЯТЬ;
│   ЕСЛИ ВВ(I+4)^='B9'Б4 ТОГДА ОПЯТЬ;
│   ЕСЛИ ВВ(I+5)^='37'Б4 ТОГДА ОПЯТЬ;
│   ЕСЛИ ВВ(I+6)^='00'Б4 ТОГДА ОПЯТЬ;
```

```
|   //---- ЕСЛИ КОД УЖЕ РАНЬШЕ ИСПРАВИЛИ - СООБЩАЕМ ОБ ЭТОМ ----
|   ЕСЛИ ВВ(I+7)='90'Б4 И ВВ(I+8)='90'Б4 ТОГДА
|      ПИСАТЬ С_НОВОЙ В_ВИДЕ('CODE ALMOST CORRECT!',СТОП);
|   ЕСЛИ ВВ(I+7)^='F7'Б4 ТОГДА ОПЯТЬ;
|   ЕСЛИ ВВ(I+8)^='F1'Б4 ТОГДА ОПЯТЬ;
|   //---- СОБСТВЕННО ИСПРАВЛЕНИЕ КОДА В ФАЙЛЕ ----
|   ВВ(I+7),ВВ(I+8)='90'Б4;
|   //---- ПРИ ЗАКРЫТИЕ ФАЙЛА ВСЕ ИСПРАВЛЕНИЯ БУДУТ СОХРАНЕНЫ ----
|   ЗАКРЫТЬ ФАЙЛ(F);
|   //---- СООБЩАЕМ ОБ УСПЕХЕ КОРРЕКЦИИ ----
|   ПИСАТЬ С_НОВОЙ В_ВИДЕ('CODE CORRECT',СТОП);
└─END;
//---- СООБЩАЕМ, ЧТО ПАКСАЛЬ-КОД ВООБЩЕ НЕ НАЙДЕН ----
ПИСАТЬ С_НОВОЙ В_ВИДЕ('^GCODE NOT FOUND !');
КОНЕЦ CORRP;
```

массив «отображенный на память» позволил представить выполняемый модуль как массив переменных БИТ(8), после чего производится поиск контекста сравнением с байтами-константами. После нахождения нужного контекста (комбинации байт) и исправления, программа просто закрывается оператором СТОП, так как механизм отображения памяти сам сохранит все изменения.

Кроме этого, работа с фалом «отображенным на память» позволяет простым способом (через общую память) организовать взаимодействие разных выполняемых модулей в среде Windows, возможно даже написанных в разных системах программирования.

Такое взаимодействие может быть организовано так: одна задача открывает файл «отображенный на память» для вывода и пишет туда в реальном времени данные (пусть строку данных). Параллельно работающая другая задача, написанная даже на другом языке, открывает этот же файл и тоже с «отображением на память», но для чтения. После чего в реальном времени читает оттуда меняющиеся данные. Чтобы не было коллизий при одновременном и чтении и записи одной и той же строки, в файле создается «кольцевой» буфер, например, на 1000 строк. По исчерпанию этого буфера первая задача начинает опять заполнять его строками сначала. В файле имеется и переменная-счетчик, показывающий номер актуальной строки. Если вторая задача не успевает с такой скорость считывать строки, она просто начнет частично пропускать их, но коллизии исчезнут, так как данные на то же место будут записываться через заведомо большой интервал (в нашем примере 1000 строк) и за это время вторая задача гарантированно успевает считывать данные из этого места без коллизий.

При подобном взаимодействии необходимо разрешить совместное использование файла «отображенного на память» нескольким задачам. В

рассматриваемой версии языка для этого используется не описатели открытия файла, а встроенная переменная с именем **?FILE_SHARE**, которую необходимо явно описать как опс ?FILE_SHARE общее бит(32); и которая задает значения трем флагам:

FILE_SHARE_READ со значением '00000001'Б4 разрешает совместное чтение;

FILE_SHARE_WRITE со значением '00000002'Б4 разрешает совместную запись;

FILE_SHARE_DELETE со значением '00000004'Б4 разрешает нескольким задачам стирать файл. Например, поставленный перед открытием файла оператор присваивания ?FILE_SHARE='00000007'Б4;

разрешает всем задачам полный доступ к файлу. Обратите внимание, что после открытия файла значение этой переменной сохраняется. Если не требуется совместный доступ для других файлов, перед операторами их открытия переменную ?FILE_SHARE необходимо обнулить.

Проверь себя

1. Как можно указать при открытии файла имя, заданное в командной строке вызова, не разбирая саму строку?
2. Если в операторах PUT/ПИСАТЬ не указывается файл, вывод по умолчанию идет на экран. Каким оператором в PL/1-программе его можно перенаправить в файл PROTOKOL.TXT?
3. С какого символа в данной версии языка начинаются имена файлов-устройств (экран, клавиатура и т.п.)?
4. Охарактеризуйте преимущества использования файла «отображенного на память».

Приложения. Список ключевых слов

ДА	= '1'B1	ПЕЧАТАТЬ	= PUT
НЕТ	= '0'B1	ВВОД	= READ
ДАТЬ_ПАМЯТЬ	= ALLOCATE	НЕ_ТЕКСТОВЫЙ	= RECORD
ВРЕМЕННОЕ	= AUTOMATIC	РЕКУРСИВНАЯ	= RECURSIVE
ОСНОВА	= BASED	ПОВТОРЯЯ	= REPEAT
БЛОК	= BEGIN	ЗАМЕНИТЬ	= REPLACE
ДВОИЧНОЕ	= BINARY	РЕСИГНАЛ	= RESIGNAL
БИТ	= BIT	ВОЗВРАТ	= RETURN
РОДНАЯ	= BUILTIN	ВОЗВРАЩАЕТ	= RETURNS
ЭТО	= BY	ОТМЕНИТЬ	= REVERT
С_ШАГОМ	= BY	ПЕРЕЗАПИСАТЬ	= REWRITE
ВЫЗВАТЬ	= CALL	ПООЧЕРЕДНЫЙ	= SEQUENTIAL
ТЕКСТ	= CHARACTER	В_УКАЗ	= SET
ЗАКРЫТЬ	= CLOSE	СИГНАЛ	= SIGNAL
СТОЛБЕЦ	= COLUMN	С_НОВОЙ	= SKIP
КОМПЛЕКСНОЕ	= COMPLEX	СТЕК	= STACK
ОПЯТЬ	= CONTINUE	ПОСТОЯННОЕ	= STATIC
С_ИМЕНАМИ	= DATA	ПАРАМЕТР	= PARAMETER
ОПС	= DCL	СТОП	= STOP
ДЕСЯТИЧНОЕ	= DECIMAL	ТЕКСТОВЫЙ	= STREAM
ОПИСАНИЕ	= DECLARE	В_СТРОКУ	= STRING
НА_МЕСТЕ	= DEFINED	ВНЕ_ПОДСТРОКИ	= STRINGRANGE
ПРЯМОЙ	= DIRECT	ИЗ_СТРОКИ	= STRING
ЦИКЛ	= DO	ВНЕ_ИНДЕКСА	= SUBSCRIPTRANGE
В_ФОРМЕ	= EDIT	ТАБУЛЯЦИЯ	= TAB
ИНАЧЕ	= ELSE	ПРОЦЕСС	= TASK
КОНЕЦ	= END	ТОГДА	= THEN
КОНЕЦ_ФАЙЛА	= ENDFILE	ПО_ИМЕНИ	= TITLE
КОНЕЦ_СТРАНИЦЫ	= ENDPAGE	ДО	= TO
ДЛЯ_ВЫЗОВА	= ENTRY	НЕТ_ФАЙЛА	= UNDEFINEDFILE
ДЛЯ_ОС	= ENVIRONMENT	ЧИСЛО_МАЛО	= UNDERFLOW
ОШИБКА	= ERROR	МАШ_КОД	= UNSPEC
ОШИБКУ	= ERROR	ДЛЯ_ИЗМЕНЕНИЙ	= UPDATE
ОБЩЕЕ	= EXTERNAL	РД	= VAR
ОБЩАЯ	= EXTERNAL	КОСВЕННОЕ	= VARIABLE
ФАЙЛ	= FILE	ЗНАЧЕНИЕ	= VALUE
ИЗ_ФАЙЛА	= FILE	РАЗНОЙ_ДЛИНЫ	= VARYING
В_ФАЙЛ	= FILE	ПОКА	= WHILE
ТОЧНОЕ	= FIXED	ВЫВОД	= WRITE
ЧИСЛО_НЕ_ПРЕДСТАВИМО	= FIXEDOVERFLOW	ДЕЛЕНИЕ_НА_0	= ZERODIVIDE
ВЕЩ	= FLOAT	--------- ВСТРОЕННЫЕ ФУНКЦИИ ------	
ВЕЩЕСТВЕННОЕ	= FLOAT	АДРЕС	= ADDR
ВВЕСТИ_ФОРМАТ	= FORMAT	ЕСТЬ_В_СТЕКЕ	= ALLOCATION
ВЕРНУТЬ_ПАМЯТЬ	= FREE	БУЛЕВА_АЛГЕБРА	= BOOL
ИЗ	= FROM	ЦЕЛОЕ_НЕ_МЕНЬШЕ	= CEIL
ЧИТАТЬ	= GET	ДАТА	= DATE
ИДТИ	= GOTO	ДАТА_4Г	= DATE4Y
ЕСЛИ	= IF	ЗАСНУТЬ	= DELAY
ЧУЖАЯ	= IMPORT	ДЕЛИТЬ	= DIVIDE
ЗАДАТЬ	= INITIAL	ОСТАТОК	= MOD
ДЛЯ_ВВОДА	= INPUT	УМНОЖИТЬ	= MULTIPLY
МЕСТНОЕ	= INTERNAL	РАЗМЕРНОСТЬ	= DIMENSION
В_ПЕРЕМ	= INTO	ЦЕЛОЕ_НЕ_БОЛЬШЕ	= FLOOR
ИНДЕКС	= KEY	УЖЕ_ОТКРЫТ	= FILEOPEN
ИНДЕКСНЫЙ	= KEYED	ПОДСЧЕТ	= TALLY
ИЗ_ИНДЕКСА	= KEYFROM	ВЕРХ_ГРАНИЦА	= HBOUND
ДЛЯ_ИНДЕКСА	= KEYTO	ИСКАТЬ	= INDEX
МЕТКА	= LABEL	НИЖ_ГРАНИЦА	= LBOUND
ХВАТИТ	= LEAVE	ДЛИНА	= LENGTH
КАК	= LIKE	ПЕЧАТАЕТСЯ_СТРОКА	= LINENO
ПЕРЕВОД_СТРОКИ	= LINE	ПУСТОЙ_УКАЗ	= NULL_PTR
С_ДЛИНОЙ_СТРОКИ	= LINESIZE	КОГДА_КОД	= ONCODE
В_ВИДЕ	= LIST	КОГДА_ФАЙЛ	= ONFILE
ГЛАВНАЯ	= MAIN	КОГДА_ИНДЕКС	= ONKEY
ПУСТО	= NULL	ПЕЧАТАЕТСЯ_СТРАНИЦА	= PAGENO
КОГДА	= ON	ОКРУГЛИТЬ	= ROUND
ОТКРЫТЬ	= OPEN	ПОДСТРОКА	= SUBSTR
ДЛЯ_ВЫВОДА	= OUTPUT	СТД_ВВОД	= SYSIN
ЧИСЛО_ВЕЛИКО	= OVERFLOW	СТД_ВЫВОД	= SYSPRINT
ПЕРЕВОД_СТРАНИЦЫ	= PAGE	ВРЕМЯ	= TIME
С_ДЛИНОЙ_СТРАНИЦЫ	= PAGESIZE	ПЕРЕВЕСТИ	= TRANSLATE
УКАЗАТЕЛЬ	= POINTER	ОЧИСТИТЬ	= TRIM
ПЕЧАТНЫЙ	= PRINT	ЦЕЛАЯ_ЧАСТЬ	= TRUNC
ДЛЯ_ПЕЧАТИ	= PRINT	ИСКАТЬ_НЕСОВПАДЕНИЕ	= VERIFY
ПРОЦ	= PROC	ЖДАТЬ	= WAIT
ПРОЦЕДУРА	= PROCEDURE	НЕ	= '^'
УКАЗ	= PTR	И	= '&'
ПИСАТЬ	= PUT	ИЛИ	= '\'

```
АДРЕС                    = ADDR
БИТ                      = BIT
БЛОК                     = BEGIN
БУЛЕВА_АЛГЕБРА           = BOOL
В_ВИДЕ                   = LIST
В_ПЕРЕМ                  = INTO
В_СТРОКУ                 = STRING
В_УКАЗ                   = SET
В_ФАЙЛ                   = FILE
В_ФОРМЕ                  = EDIT
ВВЕСТИ_ФОРМАТ            = FORMAT
ВВОД                     = READ
ВЕРНУТЬ_ПАМЯТЬ           = FREE
ВЕРХ_ГРАНИЦА             = HBOUND
ВЕЩ                      = FLOAT
ВЕЩЕСТВЕННОЕ             = FLOAT
ВНЕ_ИНДЕКСА              = SUBSCRIPTRANGE
ВНЕ_ПОДСТРОКИ            = STRINGRANGE
ВОЗВРАТ                  = RETURN
ВОЗВРАЩАЕТ               = RETURNS
ВРЕМЕННОЕ                = AUTOMATIC
ВРЕМЯ                    = TIME
ВЫВОД                    = WRITE
ВЫЗВАТЬ                  = CALL
ГЛАВНАЯ                  = MAIN
ДА                       = '1'Б
ДАТА                     = DATE
ДАТА_4Г                  = DATE4Y
ДАТЬ_ПАМЯТЬ              = ALLOCATE
ДВОИЧНОЕ                 = BINARY
ДЕЛЕНИЕ_НА_0             = ZERODIVIDE
ДЕЛИТЬ                   = DIVIDE
ДЕСЯТИЧНОЕ               = DECIMAL
ДЛИНА                    = LENGTH
ДЛЯ_ВВОДА                = INPUT
ДЛЯ_ВЫВОДА               = OUTPUT
ДЛЯ_ВЫЗОВА               = ENTRY
ДЛЯ_ИЗМЕНЕНИЙ            = UPDATE
ДЛЯ_ИНДЕКСА              = KEYTO
ДЛЯ_ОС                   = ENVIRONMENT
ДЛЯ_ПЕЧАТИ               = PRINT
ДО                       = TO
ЕСЛИ                     = IF
ЕСТЬ_В_СТЕКЕ             = ALLOCATION
ЖДАТЬ                    = WAIT
ЗАДАТЬ                   = INITIAL
ЗАКРЫТЬ                  = CLOSE
ЗАМЕНИТЬ                 = REPLACE
ЗАСНУТЬ                  = DELAY
ЗНАЧЕНИЕ                 = VALUE
И                        = '&'
ИДТИ                     = GOTO
ИЗ                       = FROM
ИЗ_ИНДЕКСА               = KEYFROM
ИЗ_СТРОКИ                = STRING
ИЗ_ФАЙЛА                 = FILE
ИЛИ                      = '|'
ИНАЧЕ                    = ELSE
ИНДЕКС                   = KEY
ИНДЕКСНЫЙ                = KEYED
ИСКАТЬ                   = INDEX
ИСКАТЬ_НЕСОВПАДЕНИЕ      = VERIFY
КАК                      = LIKE
КОГДА                    = ON
КОГДА_ИНДЕКС             = ONKEY
КОГДА_КОД                = ONCODE
КОГДА_ФАЙЛ               = ONFILE
КОМПЛЕКСНОЕ              = COMPLEX
КОНЕЦ                    = END
КОНЕЦ_СТРАНИЦЫ           = ENDPAGE
КОНЕЦ_ФАЙЛА              = ENDFILE
КОСВЕННОЕ                = VARIABLE
МАШ_КОД                  = UNSPEC
МЕСТНОЕ                  = INTERNAL
МЕТКА                    = LABEL
НА_МЕСТЕ                 = DEFINED
НЕ                       = '^'
НЕ_ТЕКСТОВЫЙ             = RECORD
НЕТ                      = '0'Б
НЕТ_ФАЙЛА                = UNDEFINEDFILE
НИЖ_ГРАНИЦА              = LBOUND
ОБЩАЯ                    = EXTERNAL
ОБЩЕЕ                    = EXTERNAL
ОКРУГЛИТЬ                = ROUND
ОПИСАНИЕ                 = DECLARE
ОПС                      = DCL
ОПЯТЬ                    = CONTINUE
ОСНОВА                   = BASED
ОСТАТОК                  = MOD
ОТКРЫТЬ                  = OPEN
ОТМЕНИТЬ                 = REVERT
ОЧИСТИТЬ                 = TRIM
ОШИБКА                   = ERROR
ОШИБКУ                   = ERROR
ПАРАМЕТР                 = PARAMETER
ПЕРЕВЕСТИ                = TRANSLATE
ПЕРЕВОД_СТРАНИЦЫ         = PAGE
ПЕРЕВОД_СТРОКИ           = LINE
ПЕРЕЗАПИСАТЬ             = REWRITE
ПЕЧАТАЕТСЯ_СТРАНИЦА       = PAGENO
ПЕЧАТАЕТСЯ_СТРОКА         = LINENO
ПЕЧАТАТЬ                 = PUT
ПЕЧАТНЫЙ                 = PRINT
ПИСАТЬ                   = PUT
ПО_ИМЕНИ                 = TITLE
ПОВТОРЯЯ                 = REPEAT
ПОДСТРОКА                = SUBSTR
ПОДСЧЕТ                  = TALLY
ПОКА                     = WHILE
ПООЧЕРЕДНЫЙ              = SEQUENTIAL
ПОСТОЯННОЕ               = STATIC
ПРОЦ                     = PROC
ПРОЦЕДУРА                = PROCEDURE
ПРОЦЕСС                  = TASK
ПРЯМОЙ                   = DIRECT
ПУСТО                    = NULL
ПУСТОЙ_УКАЗ              = NULL_PTR
РАЗМЕРНОСТЬ              = DIMENSION
РАЗНОЙ_ДЛИНЫ             = VARYING
РД                       = VAR
РЕКУРСИВНАЯ              = RECURSIVE
РЕСИГНАЛ                 = RESIGNAL
РОДНАЯ                   = BUILTIN
С_ДЛИНОЙ_СТРАНИЦЫ        = PAGESIZE
С_ДЛИНОЙ_СТРОКИ          = LINESIZE
С_ИМЕНАМИ                = DATA
С_НОВОЙ                  = SKIP
С_ШАГОМ                  = BY
СИГНАЛ                   = SIGNAL
СТД_ВВОД                 = SYSIN
СТД_ВЫВОД                = SYSPRINT
СТЕК                     = STACK
СТОЛБЕЦ                  = COLUMN
СТОП                     = STOP
ТАБУЛЯЦИЯ                = TAB
ТЕКСТ                    = CHARACTER
ТЕКСТОВЫЙ                = STREAM
ТОГДА                    = THEN
ТОЧНОЕ                   = FIXED
УЖЕ_ОТКРЫТ               = FILEOPEN
УКАЗ                     = PTR
УКАЗАТЕЛЬ                = POINTER
УМНОЖИТЬ                 = MULTIPLY
ФАЙЛ                     = FILE
ХВАТИТ                   = LEAVE
ЦЕЛАЯ_ЧАСТЬ              = TRUNC
ЦЕЛОЕ_НЕ_БОЛЬШЕ          = FLOOR
ЦЕЛОЕ_НЕ_МЕНЬШЕ          = CEIL
ЦИКЛ                     = DO
ЧИСЛО_ВЕЛИКО             = OVERFLOW
ЧИСЛО_МАЛО               = UNDERFLOW
ЧИСЛО_НЕ_ПРЕДСТАВИМО     = FIXEDOVERFLOW
ЧИТАТЬ                   = GET
ЧУЖАЯ                    = IMPORT
ЭТО                      = BY
```

Список сокращений английских ключевых слов

Ключевое слово	Сокращение	Ключевое слово	Сокращение
AUTOMATIC	AUTO	INITIAL	INIT
BINARY	BIN	INTERNAL	INT
CHARACTER	CHAR	OVERFLOW	OFL
COLUMN	COL	POINTER	PTR
COMPLEX	CPLX	PROCEDURE	PROC
CONTROLLED	CTL	STRINGRANGE	STRG
DECIMAL	DEC	SUBSCRIPTRANGE	SUBRG
DECLARE	DCL	UNDEFINEDFILE	UNDF
DEFINED	DEF	UNDERFLOW	UFL
ENVIRONMENT	ENV	VARYING	VAR
EXTERNAL	EXT	ZERODIVIDE	ZDIV
FIXEDOVERFLOW	FOFL		

Список сокращений русских ключевых слов

Ключевое слово	Сокращение
ВЕЩЕСТВЕННОЕ	ВЕЩ
ОПИСАНИЕ	ОПС
ПРОЦЕДУРА	ПРОЦ
РАЗНОЙ_ДЛИНЫ	РД
УКАЗАТЕЛЬ	УКАЗ

Предметный указатель