**Предисловие к русскому переводу**

Информация на русском языке о 8-разрядных компьютерах, операционных системах и компиляторах для них практически отсутствует. Утратив коммерческое значение они, тем не менее, представляют интерес с исторической точки зрения, отражая процесс развития программного обеспечения персональных компьютеров. Перевод Руководства программиста PL/I выполнен в 2016 году с целью сохранения компьютерного наследия, ознакомления с ранее довольно распространенным на больших ЭВМ языком программирования PL/I и как дань уважения программистам, реализовавшим его версию, работающую в очень ограниченном окружении на 8 разрядных микрокомпьютерах с 48 килобайтами ОЗУ.

В качестве исходного текста Руководства программиста PL/I для перевода на русский язык использован текст восстановленный и отформатированный в формате MS Word Брайаном Лоуренсом {Bryan Lawrence}. Некорректно восстановленные части текста и замеченные опечатки исправлены и дополнены из других руководств Digital Research. Листинги программ восстановлены из файлов исходных текстов программ, включенных в дистрибутив компилятора. Структура *Руководства программиста PL/I* полностью соответствует оригиналу. В переведенной версии руководства нумерация листингов изменена с буквенной на цифровую.

Любые торговые марки, знаки и названия программ, служб и организаций, авторские и смежные права, которые упоминаются, используются или цитируются, принадлежат их законным владельцам и их использование здесь не дает вам право на любое другое использование.

Перевод данного руководства предназначен только для свободного ознакомления с возможностями языка программирования PL/I реализованного Digital Research для 8 и 16 разрядных компьютеров с исторической точки зрения и не подразумевает вознаграждения за перевод ни в каком виде. Распространение перевода данного руководства и информации приведенной в нем с целью получения прибыли недопустимо.

Перевод руководства может содержать технические неточности и опечатки. Вы можете использовать информацию из этого документа только на свой собственный страх и риск, в качестве альтернативы используя оригинальную версию.

Язык программирования PL/I первоначально разработанный IBM хотя и не стал распространенным, продолжает использоваться до настоящего времени на современных компьютерах. Версия PL/I Digital Research в настоящее время может использоваться в нескольких эмуляторах операционной системы CP/M, на сохранившихся 8-разрядных ретро компьютерах или микрокомпьютерах подходящей архитектуры до сих пор, собираемых и поддерживаемых энтузиастами (например, DX-Designs P112 или N8VEM).

**\_ \ | / \_ \_| \_| \_\_\_| \_ \ / \ |**

**| | | / | | \_ \ \_\_| | | | / |\/ |**

**\_\_\_/ | / | \_\_| ( | | | \_\_\_/ / | |**

**\_| \_\_\_\_\_| \_/ \_\_\_| \_| \\_\_\_/ \_| \\_\_\_\_| \_| \_/ \_| \_|**

**\_| \_\_ \ \_ \ \_ \_|**

**| \_\_| \_ \ \_\_ `\_\_ \ | | | | |**

**\_\_| | ( | | | | | | \_\_ < |**

**\_| \_| \\_\_\_/ \_| \_| \_| \_\_\_\_/ \_| \\_\ \_\_\_|**

**Язык**

**PL/I**

**Руководство программиста**

Авторское право © 1983

Digital Research

P.O. Box 579

160 Central Avenue

Pacific Grove, CA 93950

(408) 649-3896

TWX 910 360 5001

Все права защищены

**Авторское право**

Авторское право 1983 Digital Research. Все права защищены. Любая часть этой публикации не может быть воспроизведена, передана, записана, сохранена в поисковой системе или переведена на любой язык или компьютерный язык, в любой форме или каким-либо образом, электронным, механическим, магнитным, оптическим, химическим, ручным или иначе, без предварительного письменного разрешения Digital Research, Post Office Box 579, Pacific Grove, California, 93950.

**Исключение ответственности**

Digital Research не дает никаких заверений или гарантий в отношении содержания данного документа, включая любые гарантии товарности или пригодности для какой-либо конкретной цели. Кроме того, Digital Research сохраняет за собой право пересмотреть эту публикацию и время от времени вносить изменения в содержание настоящего документа без обязательства Digital Research уведомить любое лицо, о таком пересмотре или изменениях.

**Торговые марки**

CP/M - зарегистрированная торговая марка Digital Research. LINK-80, LINK-86, SID, и SID-86 - торговые марки Digital Research. ADM-3A - торговая марка Lear Siegler Incorporated. IBM - товарный знак International Business Machines.

Руководство программиста языка PL/I было подготовлено, используя средство форматирования текста TEX-80 Digital Research и распечатано в Соединенных Штатах Америки. [[1]](#footnote-1)

Первое издание: сентябрь 1982

Второе издание: май 1983

Предисловие

Язык PL/I Digital Research является версией PL/I, основанной на американском национальном стандарте X3.74 подмножества PL/I общего назначения (подмножество G). Язык PL/I Digital Research представляет собой полную систему разработки программного обеспечения для приложений и системного программирования.

Digital Research реализовала PL/I для 8-разрядных и 16-разрядных микропроцессоров. На уровне исходного кода 16-разрядные версии совместимы снизу вверх с 8-разрядными версиями.

Язык PL/I Digital Research работает в любой операционной системе Digital Research. Он также работает в дисковой операционной системе версии 1.1 на персональном компьютере IBM. Это руководство предполагает, что вы уже знакомы с операционной системой, и сводит к минимуму ссылки на конкретные системы.

Руководство программиста языка PL/I является начальным учебным пособием о функциях и возможностях PL/I. Оно должно использоваться в сочетании со *Справочным руководством языка PL/I*, которое является формальной спецификацией языка, его синтаксиса и семантики.

Это руководство разделено на две части. *Первая часть*, [*Разделы 1*](#_Введение)-[*6*](#_Использование_системы), представляет краткое введение в язык PL/I, с акцентом на блочную структуру, типы данных и его различные выполняемые операторы. [*Раздел 5*](#_Стиль_программирования) дает методические рекомендации по разработке легко воспринимаемого стиля программирования. [*Раздел 6*](#_Использование_системы_1) объясняет работу системы в целом, и знакомит вас с техникой компиляции, компоновки и выполнения программы.

*Вторая часть*, [*Разделы 7*](#_Использование_различных_типов)-[*18*](#_Overlays), содержит подробные примеры программ, которые иллюстрируют полезные средства языка, включая обработку ввода-вывода, обработку строк и списков, научные расчеты и бизнес-приложения. Существует также дискуссия о прямом доступе к конкретным подпрограммам в библиотеке подпрограмм времени выполнения и написания программ использующих оверлеи. В каждом разделе представлены общие понятия, и затем подробное описание одного или нескольких примеров программ, для иллюстрации концепции.

Лучшим способом изучения любого языка программирования является изучение рабочих примеров. Чтобы освоить PL/I, вы должны изучить примеры этих программ вместе с соответствующим текстом, и сверить материал со *Справочным руководством языка PL/I* при необходимости. Как только вы понимаете работу конкретной программы, можете изменить программу, для улучшения ее работы и вашего опыта использования языка.

**Содержание**

1 Введение 1

1.1 Что такое PL/I? 1

1.2 Использование этого руководства 1

1.3 Обозначения 1

2 Язык PL/I 3

2.1 Структурные операторы 3

2.2 Операторы объявления 3

2.3 Выполняемые операторы 3

2.4 Блоки PL/I 4

2.5 Процедуры 5

2.6 DO-группы 6

3 Объявления 7

3.1 Скалярные данные 8

3.1.1 Арифметические данные 8

3.1.2 Строковые данные 9

3.1.3 Данные управления программой 10

3.1.4 Данные типа POINTER 12

3.1.5 Данные типа FILE 12

3.2 Данные агрегатов 13

3.2.1 Массивы 13

3.2.2 Структуры 14

4 Выполняемые операторы 17

4.1 Операторы присваивания 17

4.2 Операторы управления последовательностью 18

4.2.1 Циклы 18

4.2.2 Вызов процедуры 20

4.2.3 Передача параметров 20

4.2.4 Условный переход 22

4.2.5 Безусловный переход 22

4.3 Операторы ввод-вывода и обработки файла 22

4.3.1 Открытие файлов 23

4.3.2 Атрибуты файлов 25

4.3.3 Подразумеваемые атрибуты 25

4.3.4 Закрытие файлов 26

4.3.5 Методы доступа к файлу 26

4.3.6 Элементы формата данных 27

4.3.7 Элементы управления формата 27

4.3.8 Предопределенные файлы 28

4.4 Операторы обработки состояний 28

4.4.1 Оператор ON 28

4.4.2 Оператор REVERT 29

4.4.3 Оператор SIGNAL 29

4.4.4 Категории состояний 29

4.4.5 Встроенные функции обработки состояний 30

4.5 Операторы управления памятью 30

4.5.1 Переменные BASED и указатели 31

4.5.2 Оператор ALLOCATE 32

4.5.3 Оператор FREE 32

4.6 Операторы препроцессора 33

4.7 Пустой оператор 33

5 Стиль программирования 35

5.1 Регистр символов 35

5.2 Отступы 35

5.3 Сокращения 36

5.4 Модульный формат 36

5.5 Комментарии 37

6 Использование системы 39

6.1 Системные файлы PL/I 39

6.2 Вызов компилятора 40

6.3 Работа компилятора 41

6.4 Программа DEMO 43

6.5 Выполнение DEMO 44

6.6 Сообщения об ошибках и коды состояний 45

7 Использование данных различных типов 47

7.1 Программа FLTPOLY 47

7.2 Программа DECPOLY 49

8 Обработка файлов STREAM и RECORD 51

8.1 Программа копирования файла 51

8.2 Файл имен и адресов 53

8.2.1 Программа CREATE 53

8.2.2 Программа RETRIEVE 56

8.3 Система управления информацией 59

8.3.1 Программа ENTER 60

8.3.2 Программа KEYFILE 62

8.3.3 Программа UPDATE 63

8.3.4 Программа REPORT 66

9 Константы, переменные и параметры типа метки 71

9.1 Помеченные операторы 71

9.2 Метки программы 71

9.3 Вычисляемый GOTO 72

9.4 Ссылки на метку 72

9.5 Пример программы 73

10 Обработка состояний 75

10.1 Категории состояний 75

10.2 Операторы обработки состояний 75

10.2.1 Операторы ON и REVERT 75

10.2.2 Оператор SIGNAL 77

10.3 Примеры обработки состояний 77

10.3.1 Программа FLTPOLY2 77

10.3.2 Программа COPYLPT 78

11 Обработка символьной строки 83

11.1 Программа OPTIMIST 83

11.2 Функция лексического анализа 86

11.2.1 Процедура GNT 88

11.2.2 DO-группы 89

12 Обработка списков 91

12.1 Базированные переменные и указатели 91

12.2 Программа REVERSE 93

12.3 Программа сетевого планирования 96

12.3.1 Список структур NETWORK 98

12.3.2 Обход связанных списков 100

12.3.3 Общая структура программы 100

12.3.4 Процедура setup (задание) 101

12.3.5 Процедура connect 101

12.3.6 Процедура find 101

12.3.7 Процедура print\_all 101

12.3.8 Процедура print\_paths 101

12.3.9 Процедура print\_route 101

12.3.10 Процедура shortest\_distance 102

12.3.11 Процедура free\_all 102

12.3.12 Расширение программы NETWORK 102

13 Обработка рекурсии 111

13.1 Функция факториала 111

13.2 Вычисления FIXED DECINAL и FLOAT BINARY 114

13.3 Функция Аккермана 117

13.4 Механизм вычисления арифметических выражений 118

13.4.1 Процедура exp 120

13.4.2 Обработка состояния 121

13.4.3 Внесение улучшений 122

14 Раздельная компиляция 125

14.1 Объявления данных и программ 125

14.2 Данные ENTRY 126

14.3 Пример раздельной компиляции 127

15 Десятичные вычисления 133

15.1 Сравнение десятичных и двоичных операций 133

15.2 Формат представления десятичных чисел 135

15.3 Сложение и вычитание 136

15.4 Умножение 138

15.5 Деление 140

16 Коммерческие вычисления 143

16.1 Простая программа кредитования 143

16.2 Обычная рента 145

16.2.1 Смешанные типы данных 148

16.2.2 Вычисление текущей стоимости PV 149

16.2.3 Вычисление платежа PMT 150

16.2.4 Вычисление количества платежных периодов n 151

16.3 Формат графика оплаты кредита 152

16.3.1 Объявления переменных 157

16.3.2 Выполнение программы 158

16.3.3 Форматы отображения 159

16.4 Расчет графиков амортизации 164

16.4.1 Общие алгоритмы 164

16.4.2 Выбор графика 171

16.4.3 Отображение выходных данных 172

17 Подпрограммы динамического распределение памяти и стека 177

17.1 Подпрограммы динамического распределение памяти 177

17.1.1 Функции TOTWDS и MAXWDS 177

17.1.2 Подпрограмма ALLWDS 177

17.2 Функция STKSIZ 179

18 Оверлейные программы 183

18.1 Использование оверлейных программ PL/I 183

18.2 Написание оверлейных программ PL/I 184

18.2.1 Первый способ наложения 184

18.2.2 Второй способ наложения 185

18.2.3 Общие ограничения наложения 186

18.3 Синтаксис командной строки 187

**Таблицы, Рисунки и Листинги**

**Таблицы**

Таблица 3‑1 Типы данных PL/I 7

Таблица 4‑1 Допустимые файловые атрибуты 25

Таблица 4‑2 Допустимые файловые атрибуты операторов ввода-вывода 25

Таблица 4‑3 Категории состояний PL/I и подкоды 29

Таблица 6‑1 Системные файлы PL/I 40

Таблица 6‑2 Опции компилятора PL/I 41

Таблица 15‑1 Различие десятичных и двоичных данных 134

**Рисунки**

Рисунок 2‑1 Компоненты процедуры PL/I 5

Рисунок 3‑1 Массивы 13

Рисунок 3‑2 Иерархия объявления структуры 14

Рисунок 4‑1 Формы оператора DO 19

Рисунок 6‑1 Разработка программы PL/I 39

Рисунок 8‑1 Имена файлов по умолчанию в хвосте команды 52

Рисунок 18‑1 Использование оверлея в большой программе 169

Рисунок 18‑2 Древовидная структура оверлея 170

Листинги

Листинг 2‑1 Пример программы PL/I 4

Листинг 3‑1 Пример переменных меток 11

Листинг 3‑2 Внешняя процедура A 11

Листинг 3‑3 Программа CALL 12

Листинг 3‑4 Пример объявления структуры 14

Листинг 4‑1 Простой пример операторов присваивания 18

Листинг 4‑2 Передача параметров 21

Листинг 5‑1 Стилистические соглашения PL/I 36

Листинг 6‑1 Компиляция DEMO с опцией $N 43

Листинг 6‑2 Компиляция DEMO с опцией $L 43

Листинг 6‑3 Диалог с программой DEMO 44

Листинг 6‑4 Обратная трассировка ошибки в программе DEMO 44

Листинг 7‑1 Программа вычисления полинома (FLOAT BINARY) 47

Листинг 7‑2 Взаимодействие с программой FLTPOLY 48

Листинг 7‑3 Программа вычисления полинома (FIXED DECIMAL) 49

Листинг 7‑4 Взаимодействие с программой DECPOLY 50

Листинг 8‑1 Программа COPY (копирования файла) 51

Листинг 8‑2 Взаимодействие с программой COPY 52

Листинг 8‑3 Программа CREATE 53

Листинг 8‑4 Взаимодействие с программой CREATE 55

Листинг 8‑5 Вывод программы CREATE 56

Листинг 8‑6 Программа RETRIEVE 56

Листинг 8‑7 Взаимодействие с программой RETRIEVE 58

Листинг 8‑8 Программа ENTER 60

Листинг 8‑9 Взаимодействие с программой ENTER 61

Листинг 8‑10 Программа KEYFILE 63

Листинг 8‑11 Взаимодействие с программой KEYFILE 63

Листинг 8‑12 Содержимое файла ключей 63

Листинг 8‑13 Программа UPDATE 64

Листинг 8‑14 Взаимодействие с программой UPDATE 65

Листинг 8‑15 Программа REPORT 67

Листинг 8‑16 Вывод REPORT на консоль 68

Листинг 8‑17 Вывод REPORT в файл на диске 69

Листинг 9‑1 Иллюстрация переменных и констант типа меток 74

Листинг 10‑1 Программа REVERT 76

Листинг 10‑2 Программа FLTPOLY2 78

Листинг 10‑3 Программа COPYLPT 79

Листинг 10‑4 Взаимодействие с COPYLPT 81

Листинг 10‑5 Вывод программы COPYLPT 82

Листинг 11‑1 Программа OPTIMIST 84

Листинг 11‑2 Взаимодействие с программой OPTIMIST 85

Листинг 11‑3 Программа FSCAN 87

Листинг 11‑4 Взаимодействие с программой FSCAN 88

Листинг 12‑1 Программа REVERSE 93

Листинг 12‑2 Взаимодействие с программой REVERSE 94

Листинг 12‑3 Взаимодействие с программой NETWORK 96

Листинг 12‑4 Программа NETWORK 103

Листинг 13‑1 Программа IFACT 112

Листинг 13‑2 Вывод программы IFACT 113

Листинг 13‑3 Программа RFACT 113

Листинг 13‑4 Вывод программы RFACT 114

Листинг 13‑5 Программа DFACT 115

Листинг 13‑6 Вывод программы DFACT 115

Листинг 13‑7 Программа FFACT 116

Листинг 13‑8 Вывод программы FFACT 116

Листинг 13‑9 Программа ACK 118

Листинг 13‑10 взаимодействие с программой ACK 118

Листинг 13‑11 Программа EXPRESSION, использующая анализатор EXPR1 119

Листинг 13‑12 Взаимодействие с EXPR1 121

Листинг 13‑13 Анализатор выражения EXPR2 122

Листинг 13‑14 Взаимодействие с EXPR2 124

Листинг 14‑1 Иллюстрация констант и переменных ENTRY 127

Листинг 14‑2 MAININVT - основной модуль программы обращения матрицы 129

Листинг 14‑3 Подпрограмма обращения матрицы ININVT 130

Листинг 14‑4 Взаимодействие с подпрограммой INVMAT 131

Листинг 16‑1 Программа LOAN1 144

Листинг 16‑2 Вывод программы LOAN1 145

Листинг 16‑3 Программа ANNUITY 146

Листинг 16‑4 Взаимодействие с программой ANNUITY 148

Листинг 16‑5 Программа LOAN2 156

Листинг 16‑6 Первое взаимодействие с LOAN2 158

Листинг 16‑7 Второе взаимодействие с LOAN2 158

Листинг 16‑8 Третье взаимодействие с LOAN2 158

Листинг 16‑9 Четвертое взаимодействие с LOAN2 158

Листинг 16‑10 Программа DEPREC 159

Листинг 16‑11 Первое взаимодействие с DEPREC 162

Листинг 16‑12 Второе взаимодействие с DEPREC 162

Листинг 16‑13 Третье взаимодействие с DEPREC 163

Листинг 16‑14 Четвертое взаимодействие с DEPREC 164

Листинг 17‑1 Программа ALLTST 166

Листинг 17‑2 Взаимодействие с программой ALLTST 167

Листинг 17‑3 Программа ACKTST 168

Листинг 17‑4 Вывод программы ACKTST 168

# Введение

## Что такое PL/I?

Язык программирования PL/I Digital Research, можно использовать для написания приложений или программ системного уровня. Формально он основан на американском национальном стандарте X3.74 подмножестве PL/I общего назначения (подмножество G). Подмножество G имеет формальную структуру полного языка PL/I, но в некотором смысле это новый язык и во многом улучшенный по сравнению с полной версией PL/I.

Язык PL/I Digital Research прост в изучении и не сложен в использовании. Он является довольно переносимым языком, так как его дизайн обычно гарантирует аппаратную независимость. Он также более эффективен и экономически выгоден, потому что программы, написанные на PL/I проще реализовывать, документировать и поддерживать.

## Использование этого руководства

Это руководство должно помочь вам узнать язык PL/I, изучив примеры программ. Если вы ранее никогда не программировали на структурированном языке высокого уровня, таком как PL/I, сначала вы должны прочитать [*Разделы 1*](#_Введение_1)*-*[*4*](#_Выполняемые_операторы). Эти разделы содержат краткое введение в язык. Язык PL/I имеет функции, аналогичные другим языкам программирования, но он также имеет свои собственные уникальные конструкции и синтаксис.

*Разделы 1*-*4* излагают основную структуру и функциональные возможности PL/I в неформальной и концептуальной форме. Это описание поможет вам ознакомиться с полными возможностями PL/I и стимулирует вас использовать его в полную силу.

*Разделы 1*-*4* в целом не являются полным учебным руководством по программированию на языке PL/I. Если вы находите, что обзор изложен не достаточно подробно, вы можете прочитать некоторые книги, перечисленные в *Приложении 7 Справочного руководства по языку PL/I*. Вы должны также обратиться к материалу изложенному в *Разделах 1-4 Справочного руководства по языку PL/I*.

Если вы опытный программист PL/I, можете начать чтение с [*Раздела 6*](#_Использование_системы_2), описывающего компиляцию и компоновку программ.

## Обозначения

В этом документе используются следующие условные обозначения:

* Слова, написанные заглавными буквами, являются ключевыми словами PL/I или именами описываемых в тексте программ PL/I.
* Слова в нижнем регистре или в комбинации из строчных букв и цифр, разделенные символом подчеркивания представляют переменную информацию выбираемую вами. Более подробно эти слова описаны или определены в тексте.
* Примеры операторов приводятся в нижнем регистре.
* Вертикальная черта | означает альтернативу.
* ⌴ представляет собой пробел.
* Квадратные скобки [ ] включают параметры.
* Многоточие (...) свидетельствуют о том, что непосредственно предшествующий ему элемент может встретиться один, или несколько раз подряд.
* За исключением специальных символов, перечисленных выше, все остальные знаки препинания и специальные символы представляют фактическое появление этих символов.
* В тексте символ CTRL представляет управляющий символ. Таким образом, CTRL-C означает Ctrl-C. В листинге исходной программы PL/I или любом листинге, который показывает пример взаимодействия с консолью, символ представляет собой управляющий символ.
* В данном руководстве листинги программ имеют квадратные скобки на левой стороне, чтобы проиллюстрировать и подчеркнуть блочную структуру языка.
* Ссылки на материал в *Справочном руководстве языка PL/I* отмечены в конце каждого раздела, например:

***Ссылки: Справочное руководство Раздел 3.1.1***

* В этом руководстве CP/M относится к любым 8 и 16 разрядным операционным системам Digital Research. DOS относится к дисковой операционной системе персонального компьютера IBM версии 1.1.
* Распечатки примеров программ в этом руководстве используют имена устройств $CON и $LST. Это стандартно для CP/M. Однако в другой операционной системе, имена устройств могут отличаться. Это не влияет на порядок выполнения программы.
* В этом документе взаимодействие пользователя с компьютером выделено жирным шрифтом.

**Конец раздела 1**

# Язык PL/I

Каждая программа PL/I состоит из одного или нескольких операторов из следующих трех общих категорий:

* Структурные операторы;
* Операторы объявления;
* Выполняемые операторы.

Эти категории не взаимоисключающие, но обеспечивают удобную начальную точку. Следующие разделы описывают и иллюстрируют операторы каждой общей категории.

## Структурные операторы

*Структурные операторы* - основа любой программы, потому что они определяют логические модули в программе. Эти логические модули называют *блоками*. Когда программа выполняется, управление всегда переходит из одного логического модуля к другому. Логические модули могут содержать другие логические модули, заставляя управление течь в и из модулей. Структурные операторы используются для определения иерархической и логической структуры программы.

## Операторы объявления

*Операторы объявления* всегда присутствуют в логическом модуле в виде структурных операторов, и определяют среду логического модуля. *Среда* - это имена и типы всех переменных, доступных в логическом модуле. Операторы объявления используются для определения перечня переменных, которыми можно управлять в логическом модуле.

## Выполняемые операторы

*Выполняемые операторы* управляют хранением, передачей потока управления между логическими блоками, управляют потоком данных к и от устройств ввода-вывода и производят вычисления. Структурные операторы и операторы объявления служат только для создания контекста выполняемых операторов.

*Листинг 2-1* на следующей странице показывает программу PL/I, которая иллюстрирует операторы каждой категории. В этом месте вы не должны полностью понимать программу или синтаксис каждого оператора, но вы видите, что программа состоит из различных блоков операторов. Каждый блок является логическим модулем управления.

Каждая программа PL/I должна иметь блок *главной процедуры*. Несмотря на то, что вы можете отдельно разработать и скомпилировать внешние процедуры, которые можно соединить и вызвать из главной процедуры, в программе может быть только один блок главной процедуры. В *Листинге 2-1* первые два оператора, вместе с последним оператором, определяют самый верхний, или главный блок программы.

**┌── sample:**

**│ procedure options(main);**

**│ declare**

**│ c character(10) varying;**

**│**

**│ ┌── do;**

**│ │ put skip list('Input: ')**

**│ │ get list(c);**

**│ │ c = upper(c); /\* ссылка на функцию \*/**

**│ │ put skip list('Output: ',c);**

**│ └── end;**

**│**

**│ ┌── begin;**

**│ │ declare**

**│ │ c float binary(24);**

**│ │**

**│ │ put skip list('Input:**

**│ │ get list(c);**

**│ │ call output(c); /\* Вызов подпрограммы \*/**

**│ └── end;**

**│**

**│ ┌── upper:**

**│ │ procedure(c) returns(character(10) varying);**

**│ │ declare**

**│ │ c character(10) varying;**

**│ │**

**│ │ return(translate(c,'ABCDEFGHIJKLMNOPQRSTUVWXYZ',**

**│ │ 'abcdefghijklmnopqrstuvwxyz'));**

**│ └── end upper;**

**│**

**│ ┌── output:**

**│ │ procedure(c);**

**│ │ declare**

**│ │ c float binary(24);**

**│ │**

**│ │ put skip edit(c) (column(20),e(10,2));**

**│ └── end output;**

**└── end sample;**

Листинг 2‑1 Пример программы PL/I

## Блоки PL/I

В PL/I блок может иметь свою собственную локальную среду, и, возможно, среду, унаследованную от содержащего его блока. Содержащим блоком является любой блок, который включает другой блок. Например, в *Листинге 2-1* DO-группа унаследовала среду блока главной процедуры. Однако, блок BEGIN имеет свою собственную среду, несмотря на то, что он расположен в блоке главной процедуры.

В PL/I существует два типа блоков:

* блоки PROCEDURE;
* блоки BEGIN.

Вкладывать можно блоки любого типа. Это означает, что вы можете поместить один блок в другой, но блоки не могут пересекаться. Существенным различием между блоком PROCEDURE и блоком BEGIN является способ, которым PL/I выполняет каждый блок в рамках общей программы.

PL/I выполняет блоки BEGIN при встрече с ними в нормальной последовательности операторов в программе. Блок BEGIN завершается при встрече соответствующего ему оператора END или когда управление передается вне блока. Когда управление достигает блока BEGIN, операторы в блоке выполняются последовательно. Обычно, когда управление оставляет блок, оно просто передается содержащему его блоку или переходит к следующему последовательному блоку.

PL/I игнорирует блоки PROCEDURE, при встрече с ними в обычной последовательности операторов в программе. Управление передается блоку PROCEDURE только, когда программа вызывает процедуру с помощью оператора CALL или функциональной ссылки. Блок PROCEDURE активен, пока в блоке выполняются операторы. Когда операторы процедуры заканчивают выполняться, блок PROCEDURE возвращает управление в точку вызова.

Поэтому вы можете поместить процедуру в любое место программы. Хорошей практикой программирования является размещение всех процедур в нижней части главной программы. Это упрощает отладку и сопровождение программы.

## Процедуры

Каждая процедура состоит из имени процедуры, заголовка процедуры, тела процедуры нуля или большего количества операторов и завершающего оператора. *Рисунок 2-1* на примере программы показывает компоненты главной процедуры.

┌─────────────────────┐

│ Заголовок процедуры │

└──────────┬──────────┘

┌───────────────┐ │

│ Имя процедуры ├─────────> **sample:** V

└───────────────┘ **procedure options(main);**

┌────────────────┐ ┌ **.**

│ Тело процедуры ├─────────────> ┤ **.**

└────────────────┘ └ **.**

┌──────────────────────┐

│ Завершающий оператор ├──> **end sample;**

└──────────────────────┘

Рисунок 2‑1 Компоненты процедуры PL/I

Если вы вкладываете процедуры, они наследуют среду содержащих блоков. Однако любая переменная, которую вы объявляете в содержащем блоке, может быть повторно объявлена во вложенной процедуре с локальными атрибутами.

В PL/I существуют два типа общих процедур:

* подпрограммы;
* функции.

Вы используете оператор CALL, чтобы вызвать процедуру подпрограммы. Процедура подпрограммы выполняет определенную задачу, и необязательно возвращает значения процедуре вызова.

Вы можете вызвать процедуру функцию, делая ссылку на функцию. Функциональная ссылка просто использует имя функции в операторе. PL/I, вычисляет указанную функцию и заменяет ее скалярной величиной в месте расположения ссылки.

Процедуры бывают внутренние или внешние относительно главной процедуры. *Внутренняя процедура* содержится в теле главной процедуры, в то время, как *внешние процедуры* пишутся и компилируются отдельно от главной программы. Чтобы сделать внешнюю процедуру известной главной процедуре, вы должны объявить имя процедуры в качестве константы типа ENTRY (см. [*Раздел 3.1.3*](#_Данные_управления_программой)). Также вы должны соединить внешнюю процедуру с главной процедурой после того, как обе будут скомпилированы. В программе sample все процедуры внутренние по отношению к главной процедуре.

## DO-группы

DO-группа подобна блоку BEGIN. Есть несколько форм DO-групп, и они являются выполняемыми операторами, так как они влияют на поток управления. Однако их также считают структурными операторами, потому что они определяют логические блоки.

*Листинг 2-1* иллюстрирует самую простую форму DO-группы. Она похожа на блок BEGIN, но имеет существенное отличие. Несмотря на то, что DO-группа связывает все операторы в своем теле в один логический блок, она не может определить новую среду. DO-группа не может определять новые переменные, среда которых ограничена телом DO-группы.

DO-группа может связать только исполнимые операторы. Между тем, блок BEGIN может связать операторы описания и исполнимые операторы. Среда DO-группа определяется средой блока, в котором на находится.

Ссылки: Справочное руководство Разделы с 2.1 по 2.19, с 8.1 по 8.2

**Конец раздела 2**

# Объявления

Вы можете использовать операторы объявления для определения элементов данных которыми вы хотите управлять с помощью выполняемых операторов в вашей программе. PL/I имеет много разных типов данных. В дополнение к арифметическим и строковым данным, PL/I поддерживает объекты данных указателей, меток и точек входа (entry), которые обычно не доступны в других языках. *Таблица 3-1* показывает типы данных PL/I, разделенные на категории и подкатегории.

Таблица 3‑1 Типы данных PL/I

|  |  |
| --- | --- |
| **Категория** | **Подкатегория** |
| **Арифметическая** | FIXED BINARY |
| FLOAT BINARY |
| FIXED DECIMAL |
| **Строковая** | CHARACTER |
| BIT |
| **Управления** | Переменная типа меткки |
| Константа типа метки |
| Переменная типа Entry |
| Константа типа Entry |
| **Указатель** | POINTER |
| **Файл** | Файдлвая переменная |
| Файловая константа |
| **Данные агрегатов** | Массивы |
| Структуры |
| **Процедуры** | Подпрограммы |
| Функции |

Операторы объявления должны определять все данные констант или переменных. Вы должны явно объявить все данные переменных в операторе DECLARE, но данные *констант* обычно объявляются неявно, появлением их в выполняемом операторе. *Переменная* PL/I определяется именем идентификатора. *Имя* может состоять максимум из тридцать одного алфавитно-цифрового символа или символа подчеркивания. Первый символ должен быть буквой.

Обычно, операторы объявления явные или неявные, приводят к определенному выделению хранения (области памяти) для объявленного элемента данных. Компилятор определяет объем памяти, требуемый для типа данных, и связывает элемент с этим хранением. Базированные переменные BASED являются исключением, потому что они не обязательно вызывают выделение памяти (см. [*Раздел 4.5*](#_Операторы_управления_памятью)).

## Скалярные данные

Существуют два основных вида данных: *скаляр* или один элемент данных, и *агрегат* или несколько элементов данных. Скалярные типы данных являются основными типами данных языка.

### Арифметические данные

Вы можете использовать арифметические данные для прямых численных расчетов. PL/I обеспечивает несколько типов арифметических данных, таким образом, вы можете использовать данные подходящие приложению.

#### FIXED BINARY

Вы можете использовать данные FIXED BINARY, для представления целых чисел. PL/I внутренне представляет этот тип данных в дополнительном двоичном виде. Точность числа FIXED BINARY - количество битов, используемых для его представления, независимо от знака. PL/I использует от 1 до 15 битов, таким образом, он может представлять целые числа в диапазоне от -32768 до +32767.

#### FLOAT BINARY

Вы можете использовать данные FLOAT BINARY для представления очень маленьких или очень больших чисел. Данные FLOAT BINARY имеют двоичную дробная часть (называемую мантиссой), двоичный порядок и знак. PL/I поддерживает числа FLOAT BINARY с одинарной и двойной точностью. Точностью числа FLOAT BINARY является количество битов в мантиссе.

Числа *одинарной точности* могут иметь от 1 до 24 битов, тогда как часть экспоненты всегда представлена 8 битами. Числа FLOAT BINARY одинарной точности в десятичном представлении могут иметь максимальный диапазон приблизительно от 10-39 до 1038.

Числа *двойной точности* могут иметь от 24 до 53 битов, в то время как экспонента имеет 11 битов. Максимальный диапазон чисел FLOAT BINARY двойной точности в десятичном представлении - приблизительно от 10-308 до 10308.

#### FIXED DECIMAL

Вы можете использовать данные FIXED DECIMAL для представления чисел с фиксированной десятичной точкой. Также вы можете использовать данные FIXED DECIMAL для представления *целых чисел*. Внутренне, PL/I представляет данные FIXED DECIMAL в виде двоично-десятичных (BCD[[2]](#footnote-2)) цифр.

Числа FIXED DECIMAL имеют точность и масштабный коэффициент. *Точность* - общее количество десятичных цифр, используемых для представления числа. *Масштабный коэффициент* определяет количество десятичных цифр справа от десятичной точки.

В PL/I точность числа FIXED DECIMAL может измениться от единицы до пятнадцати, в то время как масштабный коэффициент может измениться от нуля до пятнадцати. Этот арифметический тип данных особенно полезен для коммерческих расчетов, которые требуют точного представления долларов и центов и не подвержен ошибкам усечения двоичной арифметики.

Вы можете объявлять переменные арифметических данных в операторе объявления в одной из следующих форм, в которых p - точность, и q - масштабный коэффициент.

Оператор:

**DECLARE идентификатор FIXED BINARY [(p)];**

Пример:

**declare index\_counter fixed binary(7);**

Оператор:

**DECLARE идентификатор FLOAT BINARY[(p)];**

Пример:

**declare pi float binary(53);**

Оператор:

**DECLARE идентификатор FIXED DECIMAL [(p[,q])];**

Пример:

**declare base\_pay fixed decimal(5,2);**

**Примечание**: точность и масштабный коэффициент - необязательные. Если вы опускаете их, PL/I использует значения по умолчанию.

Вы должны использовать двоичную арифметику для большинства численных расчетов, потому что она быстрее и использует меньше памяти. Если вы занимаетесь научной работой, PL/I предоставляет вам полную библиотеку встроенных математических функций, которая включает тригонометрическое и гиперболические функции.

### Строковые данные

Возможность управлять строковыми переменными является одной из самых полезных функций PL/I. PL/I имеет полный набор встроенных функций, которые вы можете использовать для управления строковыми данными. Вы объявляете строковую переменную в качестве битовой или символьной строки в объявлении одной из следующих форм:

Оператор:

**DECLARE идентификатор CHARACTER[(n)];**

Пример:

**declare alphabet character(26);**

Оператор:

**DECLARE идентификатор CHARACTER[(n)] VARYING;**

Пример:

**declare state character(20) varying;**

Оператор:

**DECLARE идентификатор BIT[(n)];**

Пример:

**declare flag bit(1);**

Атрибут VARYING означает, что символьная строка может измениться по длине, но не может превысить значение n. Для переменных CHARACTER значение n может быть между 0 и 254. Если вы хотите управлять более длинными строками, вы можете использовать одномерные массивы.

Символьные строковые константы неявно объявляются появлением их в программе. Вы указываете символьную строковую константу, заключая ее в одиночные апострофы. Для включения апострофа в строку, вы должны перед ним поставить дополнительный апостроф. Также PL/I позволяет включать в символьную строку управляющие символы.

Ниже приводятся примеры символьных строк:

**'Ada Lovelace'  
'^g ^g Input Error'  
'Can''t Read Previous Line'**

Переменные битовых строк не могут иметь атрибут VARYING, и максимальная длина битовой строки не может превышать шестнадцать разрядов. PL/I позволяет вам определять константы строки битов в нескольких различных форматах. Каждый формат соответствует некоторому основанию, которое является числом битов, используемых, для представления элемента. Форматы для констант строки битов следующие:

* основание 1 (формат **B** или **B1**)
* основание 4 (формат **B2**)
* основание 8 (формат **B3**)
* основание 16 (формат **B4**)

В каждом из форматов вы записываете константу битовой строки, как строку числовых цифр для желаемого основания, заключенную в одиночные апострофы за которой следует тип формата. Ниже приводятся примера четырех форматов:

'101111'B эквивалентно 101111  
'101111'B1 эквивалентно 101111  
'233'B2 эквивалентно 101111  
'57'B3 эквивалентно 101111  
'2F'B4 эквивалентно 00101111

### Данные управления программой

Существует два типа данных управления программой:

* метки (LABEL);
* процедуры (ENTRY).

Данные LABEL позволяют вам ссылаться на отдельные операторы в своей программе. PL/I не только позволяет отдельным операторам иметь метки, он также позволяет вам объявлять переменные типа метки. Это означает, что вы можете управлять метками в своей программе аналогично любым другим допустимым элементом данных.

Значение переменной метки всегда - константа метки, неявно определенная и объявленная ее присутствием в качестве метки оператора в программе. PL/I позволяет вам индексировать константы типа метки. Также, вы можете объявить массивы переменных типа метки.

Вы можете использовать переменные метки, чтобы управлять потоком управления между логическими блоками программы. Рекомендуется осуществлять переходы, не используя GOTO и метки.

Следующая программа является бесполезным примером использования переменных типа меток.

**┌── chase\_your\_tail:**

**│ procedure options(main);**

**│ declare**

**│ wherever label;**

**│**

**│ there:**

**│ wherever = here;**

**│ here:**

**│ wherever = there;**

**│ goto wherever;**

**│**

**└── end chase\_your\_tail;**

Листинг 3‑1 Пример переменных меток

PL/I также поддерживает мощный тип данных называемый данные ENTRY. Данные ENTRY позволяют вам ссылаться на процедуры точно так же, как на любой допустимый элемент данных. Вы можете объявить переменную в качестве ENTRY, затем присвоить ей значение. Значением переменной ENTRY является константа ENTRY.

Константами ENTRY являются метки процедур, а не метки исполняемых операторов. Константа ENTRY неявно объявляется при появлении ее в качестве метки внутренней процедуры.

Когда вы объявляете переменную ENTRY, вы должны явно определить тип константы ENTRY, которую можно присвоить переменной. Когда вы явно объявляете константу ENTRY, вы должны объявить ее с теми же атрибутами, что у процедуры, на которую она ссылается.

Модули, показанные в *Листингах 3-2* и *3-3*, иллюстрируют эти понятия. *Листинг 3-2* содержит внешнюю процедуру, названную a. *Листинг 3-3* показывает программу CALL, которая ссылается на a. В программе CALL f - переменная ENTRY, которая принимает три различных значения констант. Для создания программы, вы компилируете каждый модуль отдельно, затем соединяете их.

┌── **a:**

│  **procedure(x) returns(float); /\* внешняя процедура \*/**

│  **declare x float;**

│  **return(x/2);**

└── **end a;**

Листинг 3‑2 Внешняя процедура A

┌── **call:**

│  **procedure options(main);**

│  **declare**

│  **f(3) entry(float) returns(float) variable,**

│  **a entry(float) returns(float); /\* константа entry \*/**

│  **declare**

│  **i fixed, x float;**

│

│  **f(1) a;**

│  **f(2) b;**

│  **f(3) c;**

│

│ ┌── **do i = 1 to 3;**

│ │ **put skip list('Type X')**

│ │ **get list(x);**

│ │ **put list('f(',i,')=',f(i)(x));**

│ └── **end;**

│  **stop;**

│

│ ┌── **b:**

│ │ **procedure(x) returns(float); /\* внутренняя процедура \*/**

│ │ **declare x float;**

│ │ **return (2\*x + 1);**

│ └── **end b;**

│

│ ┌── **c:**

│ │ **procedure(x) returns(float); /\* внутренняя процедура \*/**

│ │ **declare x float;**

│ │ **return(sin(x));**

│ └── **end c;**

│

└── **end call;**

Листинг 3‑3 Программа CALL

### Данные типа POINTER

Данные указателя позволяют вам управлять хранением, выделенным переменным. Значение переменной указателя является адресом другой переменной.

### Данные типа FILE

Элементы данных файловой переменной описывают и обеспечивают доступ к данным, связанным с внешним устройством. Элементами данных файла являются файловые константы или файловые переменные. Вы должны всегда присваивать файловой переменной файловую константу, прежде чем получите доступ к данным в файле.

Вы можете объявлять данные файла в операторе объявления в одной из следующих форм:

Оператор:

**DECLARE идентификатор FILE;**

Пример:

**declare current\_transaction file;**

Оператор:

**DECLARE идентификатор FILE VARIABLE;**

Пример:

**declare f(2) file variable;**

Выполняемые операторы, используемые для доступа к файлу, определяют атрибуты файла. ([*Раздел 4.3*](#_Операторы_ввод-вывода_и) описывает обработку файлов и операции ввода-вывода.)

## Данные агрегатов

Агрегат данных представляет собой комбинацию типов данных, которая формирует тип данных на более высоком уровне. В PL/I есть два вида агрегатов:

* массивы;
* структуры.

### Массивы

Массив представляет собой индексируемую совокупность элементов данных одного и того же типа. PL/I допускает массивы арифметических значений, символьных строк, битовых строк, констант типа меток, констант точек входа (entry), указателей, файлов или структур (см. [*Раздел 3.2.2*](#_Структуры)).

Ниже приводятся примеры объявлений массива:

**declare test\_scores(100);  
declare A(4,5);  
declare A(1:4,2:5,0:10);**

Вы можете делать прямые ссылки на отдельные элементы массива при помощи ссылок индексированной переменной. Также PL/I позволяет вам делать ссылки на сечения массива с ограничением, что ссылка должна определять компонент данных, хранение которого непрерывно. Например, используя следующие объявления:

**declare A(5,2) fixed binary;  
declare B(5,2) fixed binary;**

можете представить массивы, изображенные на *Рисунке 3-1*:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **A** | |  | **B** | |
|  | 1 | 2 |  | 1 | 2 |
| 1 |  |  | 1 |  |  |
| 2 |  |  | 2 |  |  |
| 3 |  |  | 3 |  |  |
| 4 |  |  | 4 |  |  |
| 5 |  |  | 5 |  |  |

Рисунок 3‑1 Массивы

В этом примере А и В имеют одинаковые размеры и атрибуты. Таким образом, присвоение

**A(3) = B(4);**

допустимо, потому что ссылка на сечение определяет непрерывное хранение.

### Структуры

Структура - сильно отличающийся от массива тип агрегированных данных. Структура является иерархической, как у дерева, где листья дерева, называемые узлами, могут иметь различные типы данных PL/I.

Каждый узел дерева, начиная с корня, имеет имя и номер уровня. Номер уровня указывает уровень каждого узла относительно корня. Следующий пример иллюстрирует описание структуры.

**declare**

**1 employee,**

**2 name\_address,**

**3 name,**

**4 first character(10),**

**4 middle\_initial character(1),**

**4 last character(20),**

**3 address,**

**4 street character(40),**

**4 city character(10),**

**4 state character(2),**

**4 zip\_code character(5),**

**2 position,**

**3 department\_no character(3),**

**3 job\_title character(20),**

**2 salary fixed decimal(8,2),**

**2 number\_of\_dependents fixed,**

**2 health\_plan bit(1),**

**2 date\_hired character(B);**

Листинг 3‑4 Пример объявления структуры

*Рисунок 3-2* иллюстрирует иерархию уровней, соответствующую объявлению.

employee ──┬── **name\_address** ──────┬── **name** ─────────┬── **first**

│ │ ├── middle\_initial

│ │ └── **last**

│└── **address** ──────┬── **street**

│├── **city**

├── position ──────────┬── **department\_no** ├── **state**

│└── **job\_title** └── **zip\_code**

├── **salary**

├── **number\_of\_dependents**

├── health\_plan

└── date\_hired

Рисунок 3‑2 Иерархия объявления структуры

Узлы на каждом уровне могут также определить структуру. Такая подструктура является членом основной структуры. Вы можете задать атрибут BASED основной структуры, так что в итоге все члены структуры получают атрибут.

Структуры являются мощным инструментом, так как они позволяют вам сгруппировать логически связанные элементы данных различных типов. Таким образом, структуры позволяют вам характеризовать и управлять логическими объектами в своей программе, наиболее близко отражающие реальные данные.

Ссылки: Справочное руководство разделы с 3.1 по 3.6, с 5.1 по 5.5

**Конец раздела 3**

# Выполняемые операторы

Категория выполняемых операторов делится на несколько подкатегорий на основе типа функции, которую выполняет оператор. Подкатегории следующие:

* операторы присваивания;
* операторы управления последовательностью выполнения;
* ввода-вывода и обработки файлов;
* операторы управления памятью;
* условные операторы;
* операторы препроцессора;
* пустой оператор.

## Операторы присваивания

Оператор *присваивания* помещает значение выражения в место хранения, связанное с переменной.

*Выражение* - комбинация операндов, знаков операции, функциональных ссылок и круглых скобок, которые управляют порядком вычисления выражения.

В PL/I оператор присваивания имеет форму:

**ссылка\_на\_переменную = выражение;**

Выражение в PL/I может быть довольно сложным. Самый простой тип ссылки на переменную инстанцирование имени переменной, что означает присвоение переменной определенного значения. Ссылка на переменную может также быть ссылкой на агрегат данных, или на компонент агрегата. Если переменная базированная, требуется ссылка однозначно определенного указателя (см. [*Раздел 4.5.1*](#_Переменные_BASED_и)*)*.

PL/I также позволяет в качестве цели появляться на левой стороне операторов присваивания некоторым встроенным функциям, таким как UNSPEC и SUBSTR. Когда они появляются в качестве переменных, в этом контексте их называют *псевдопеременными*.

Выражения могут быть *вычислительными*. Это означает, что выражение включает арифметические или строковые значения различных типов и соответствующие им операторы. Выражения могут также быть *невычислительными*, включающие сравнение невычислительных типов данных, таких как метки, константы entry и указатели.

PL/I позволяет вычислительные выражения различных типов данных, и автоматически выполняет преобразование между различными типами следуя стандартному набору правил по умолчанию. Вы должны ознакомиться с правилами автоматических преобразований и свойствами встроенных функций преобразования (см. *Раздел 4 Справочного руководства PL/I*).

Следующая последовательность кода иллюстрирует некоторые простые примеры операторов присваивания. Эти примеры также иллюстрируют некоторые способы, которыми вы можете сослаться на переменную в PL/I. Такие ссылки могут также произойти в выражениях, несмотря на то, что PL/I ограничивает выражения с агрегатами сравнением на равенство.

**assign:**

**procedure options(main);**

**declare**

**p pointer,**

**i fixed binary(7),**

**r bit(16),**

**s bit(16) based,**

**(u,v) float binary(24),**

**A(5,2) character(2),**

**B(5,2) character(2),**

**C character(20),**

**1 W,**

**2 x fixed binary,**

**2 y bit(16),**

**1 Z,**

**2 x fixed binary,**

**2 y bit(16);**

**.**

**.**

**.**

**u = u + v; /\* простое присваивание \*/**

**A = B; /\* присваивание всего массива \*/**

**A(3) = B(3); /\* ссылка на сечение \*/**

**w = z; /\* присвоение всей структуры \*/**

**p -> s = (r = w.y); /\* ссылка определяемая указателем \*/**

**w.x = w.x + z.x; /\* ссылка на частично определенный агрегат \*/**

**unspec(w.y) = unspec(A(5,1)); /\* ссылка на псевдопеременную \*/**

**substr(C,i+1,3) = substr(C,10,3); /\* ссылка на псевдопеременную \*/**

**A(2\*i+1) = B(4); /\* переменная является выражением \*/**

**.**

**.**

**.**

**end assign;**

Листинг 4‑1 Простой пример операторов присваивания

## Операторы управления последовательностью

Вы можете использовать операторы управления последовательностью, чтобы изменить нормальный последовательный поток управления. В PL/I операторы управления последовательностью выполняют безусловный переход, условный переход, циклы, переход и возврат с помощью вызова процедуры, и более уникальную конструкцию называемую обработчиком состояния.

### Циклы

PL/I обеспечивает большое разнообразие итеративного управления в различных формах оператора DO. Например, вы можете выполнить итерацию не только с арифметической переменной управления, но также и с помощью переменной управления указателем, которая перемещается по связанному списку указателей.

Следующие диаграммы иллюстрируют основные формы оператора DO и поток управления, который они вызывают. Значения e1, е2, е3 и е4 представляют любые скалярные выражения.

|  |  |
| --- | --- |
| **DO;**  **...**  **END;** |  |
| **DO WHILE(E);**  **...**  **END;** |  |
| **DO i = e1 REPEAT(e2);**  **...**  **END;** |  |
| **DO i = e1 REPEAT(e2) WHILE(e3);**  **...**  **END;** |  |

Рисунок 4‑1 Формы оператора DO

|  |  |
| --- | --- |
| **DO e1 TO e2 BY e3 WHILE(e4);**  **...**  **END;** |  |

**Рисунок 4 1 Формы оператора DO (продолжение)**

### Вызов процедуры

Переход и возврат происходят как результат вызова процедуры. Как мы уже видели, в PL/I есть два типа процедур: процедуры подпрограммы и процедуры функции. Есть две соответствующие формы вызова.

Вы вызываете процедуру подпрограммы с помощью оператора CALL, но вы вызываете процедуру функцию с помощью ее имени в выражении. Вы вызываете подпрограммы процедуры по определенной причине, например, изменение значений переменных, переданных в процедуру или ввода и вывода. Вы всегда вызываете функциональную процедуру в выражении, чтобы возвратить скалярный элемент данных. В PL/I оба типа процедур могут быть *рекурсивными*, что означает, что они могут вызвать себя.

Существует важное различие между определением процедуры и вызовом процедуры. Определением процедуры является оператор объявления. Вызов процедуры - это выполняемый оператор. Элементы данных, которые вы передаете процедуре при ее вызове, называют *аргументами*. Аргументы отличаются от параметров, которые вы задаете в определении процедуры. Таким образом аргументы являются параметрами, как они известны в блоке вызова, в то время как *параметры* являются соответствующими параметрами, как они известны в вызываемом блоке процедуры.

### Передача параметров

В PL/I можно передать параметры ссылкой или значением. Вы передаете их *ссылкой*, если аргументы и параметры совместно используют хранение. Вы передаете их *значением*, если значение параметра сохраняется как локальная копия значения аргумента.

По правилам PL/I параметр и аргумент всегда совместно используют хранение, если они имеют идентичные атрибуты. Если аргумент является выражением, или если атрибуты его данных не соответствуют соответствующим параметрам, то параметр передается значением. PL/I передает параметр значением, если вы заключаете параметр в круглые скобки в операторе заголовка процедуры при определении процедуры.

Процедура является независимым логическим блоком, который выполняет определенную функцию. Если вы тщательно определяете конкретную функцию, выполняемую процедурой и параметры, которые она ожидает от среды вызова, вы можете разделить проектирование, кодирование и отладку всей программы на отдельные модули.

При передаче параметра ссылкой для экономии хранения, следует знать, что вызванная процедура может изменить значение переменной вне ее среды. Если вы хотите гарантировать, что процедура не изменяет переменную вне своей среды, то вы должны передавать параметры значением и использовать дополнительное место для хранения.

Передача параметров является компромиссом между объемом доступной памяти в системе и уровнем модульности и изоляции, которую вы хотите в своей программе. Есть три варианта передачи параметров, характеризуемые как затратный, средний и экономный метода. Макет программы в *Листинге 4-1* иллюстрирует понятия, которые они представляют.

**┌── main:**

**│ procedure options(main);**

**│ declare**

**│ a float binary;**

**│ . . .**

**│ call low\_sub(a); /\* передача ссылкой \*/**

**│ call middle\_sub((a)); /\* передача значением \*/**

**│ call high\_sub(a); /\* передача ссылкой \*/**

**│ . . .**

**│ ┌── low\_sub:**

**│ │ procedure(x);**

**│ │ declare**

**│ │ x float binary;**

**│ │ . . .**

**│ └── end low\_sub;**

**│**

**│ ┌── middle\_sub:**

**│ │ procedure(x);**

**│ │ declare**

**│ │ x float binary;**

**│ │ . . .**

**│ └── end middle-sub;**

**│**

**│ ┌── high\_sub:**

**│ │ procedure(x);**

**│ │ declare**

**│ │ (x,my\_x) float binary;**

**│ │ my\_x = x; /\* переприсвоение локальной переменной \*/**

**│ │ . . .**

**│ └── end high\_sub;**

**└── end main;**

Листинг 4‑2 Передача параметров

При *экономном* методе вы передаете параметры ссылкой, но обращаете пристальное внимание на возможные побочные эффекты, которые могут возникнуть. Преимущество этого метода состоит в том, что он экономит память.

При *среднем* методе вы передаете параметры значением, заключая параметр в круглые скобки при вызове в операторе CALL или функциональной ссылке.

При *затратном* методе вы объявляете дубликат переменной для каждого параметра в определении процедуры. Затем вы присваиваете соответствующий параметр его копии и используете дубликат в качестве локальной копии в процедуре. Точно так же вы можете заключить параметр в круглые скобки в заголовке процедуры. Затратный метод является наименее эффективным в использовании памяти.

### Условный переход

PL/I обеспечивает условный переход в форме оператора IF. Условный перехода имеет одну из следующих форм:

**IF условие THEN группа  
IF условие THEN группа\_1 ELSE группа\_2**

где условие - скалярное выражение, которое PL/I вычисляет и сводит до одного значения, и группа - один оператор, DO-группы или блоки BEGIN.

Вы можете вложить операторы IF, в этом случае PL/I сопоставляет каждый ELSE с самой внутренней несопоставленной парой IF-THEN. Однако, вы можете использовать пустые операторы после ELSE, чтобы вызвать произвольное соответствие операторов ELSE с парами IF-THEN. (См. [*Раздел 4.7, "Пустой оператор"*](#_Пустой_оператор).)

### Безусловный переход

PL/I обеспечивает безусловный переход в форме оператора GOTO. Безусловный переход имеет одну из форм:

**GOTO константа\_типа\_метки;  
GOTO переменная\_типа\_метки;**

Поскольку язык PL/I является блочно-структурированным, к использованию GOTO применяются определенные правила. Целевая метка должна находиться в блоке, содержащем GOTO, или в содержащем его блоке. Вы не можете передать управление внутреннему блоку.

## Операторы ввод-вывода и обработки файла

Выполняемые операторы ввода-вывода предоставляют PL/I независимую от устройств систему ввода-вывода, которая позволяет программе передавать данные между памятью и внешними устройствами. Чтобы понять операторы ввода-вывода, вы должны сначала узнать о файлах и их атрибутах.

Набор элементов данных, которые вы передаете в или из внешнего устройства, называют *набором данных*. Соответствующую внутреннюю файловую константу или переменную, называют *файлом*.

Как и другие элементы данных, вы должны объявить файлы, прежде чем использовать их в программе. Объявление файла имеет форму:

**DECLARE file\_id FILE [VARIABLE];**

где file\_id - идентификатор файла. Если вы не включаете необязательный атрибут VARIABLE, объявление определяет *файловую константу*. С атрибутом VARIABLE объявление определяет *файловую переменную*, которой можно присвоить значение файловой константы с помощью оператора присваивания. Вы должны присвоить файловой переменной файловую константу, прежде чем сможете выполнить любые операции ввода-вывода.

### Открытие файлов

PL/I требует, чтобы файл был открыт прежде, чем выполнить любые операции ввода-вывода с набором данных. Вы можете открыть файл явно при помощи оператора OPEN, или неявно, получив доступ к файлу с помощью следующих операторов ввода-вывода:

* GET EDIT
* PUT EDIT
* GET LIST
* PUT LIST
* READ
* WRITE
* READ Varying
* WRITE Varying

Общая форма оператора OPEN

**OPEN FILE(file\_id) [атрибуты файла];**

где file\_id - идентификатор файла, который появляется в операторе объявления FILE, и атрибуты файла обозначают один или несколько из следующих:

|  |  |
| --- | --- |
| * STREAM RECORD * PRINT * INPUT/OUTPUT/UPDATE * SEQUENTIAL/DIRECT * KEYED | * TITLE * ENVIRONMENT * PAGESIZE * LINESIZE |

Несколько атрибутов в одной строке могут конфликтовать между собой, поэтому, вы можете определить только один атрибут. Первый перечисленный атрибут является *атрибутом по умолчанию*. Все атрибуты являются необязательными. Вы можете определять их в любом порядке.

Файл STREAM (потоковый) содержит данные ASCII переменной длины. Вы можете представить его себе как поток данных символов ASCII, организованных в строки и страницы. Каждая строка в файле STREAM определяется *маркером строки*, который является символом перевода строки или парой символов возврата каретки и перевода строки. Каждая страница определяется *маркером страницы*, который является символом прогона страницы. Обычно вы должны преобразовать данные файла STREAM из символьной формы в чисто двоичную форму перед их использованием. Некоторые текстовые редакторы автоматически вставляют перевод строки после каждого возврата каретки, но файлы, которые создает PL/I, могут содержать переводы строки, без предшествующего возврата каретки. В этом случае PL/I обнаруживает конец строки, когда он встречается с переводом строки.

Файл RECORD (записеориентированный) содержит двоичные данные. PL/I получает доступ к данным в блоках, определенных объявленным размером записи, или используемым вами размером элемента данных, для получения доступа к файлу. Файл RECORD должен также иметь атрибут KEYED, если вы используете ключи FIXED BINARY для получения доступа непосредственно к записям фиксированной длины.

Атрибут PRINT применяется только к файлам STREAM OUTPUT (потоковым выходным). PRINT указывает, что данные предназначены для вывода на построчный принтер.

Для файла INPUT (входного) PL/I предполагает, что файл уже существует, когда он выполняет оператор OPEN. Когда он выполняет оператор OPEN для файла OUTPUT (выходного), PL/I при этом создает файл. Если файл уже существует, PL/I сначала удаляет его, затем создает новый.

Вы можете читать из и писать в файл UPDATE (обновляемый). PL/I создает файл UPDATE, если он не существует при выполнении оператора OPEN. Файл UPDATE не может иметь атрибут STREAM.

Вы получаете доступ к файлам SEQUENTIAL (последовательным) последовательно с начала до конца, но вы получаете доступ к файлам DIRECT (прямого доступа), в произвольном порядке используя ключи. PL/I автоматически присваивает файлам DIRECT атрибут RECORD. PL/I также требует, чтобы вы объявили все файлы UPDATE с атрибутом DIRECT, таким образом, вы можете определить местоположение отдельных записей.

Файл KEYED является простым файлом с записями фиксированной длины. Ключ представляет собой относительную позицию записи в файле, основанную на записях фиксированного размера. Вы должны использовать ключи для получения доступа к файлу KEYED. PL/I автоматически назначает файлу KEYED атрибут RECORD.

Атрибут TITLE(c) определяет программируемую связь между внутренним именем файла и внешним устройством или файлом в файловой системе операционной системы. Если вы опускаете атрибут TITLE, PL/I присваивает по умолчанию file\_id.DAT, где file\_id - идентификатор файла, определенный в операторе OPEN.

Символьная строка c может определить физическое устройство, такое как консоль или принтер (см. *Раздел 10.2 в Справочном руководстве PL/I*). Если символьная строка c определяет дисковый файл, она должна быть в форме:

**d:имя\_файла.тип;пароль**

где d - код диска, тип файла и пароль являются необязательными. Вы должны определить имя файла. Имя файла не может быть неоднозначной ссылкой с подстановочными знаками.

Вы можете также определить 1$ или 2$ и для имени файла и для типа файла. 1$ получает первое имя по умолчанию из командной строки, 2$ получает второе имя по умолчанию.

Атрибут ENVIRONMENT определяет размеры фиксированных записей для записеориентированных (с атрибутом RECORD) файлов, размеры внутренних буферов, режим открытия файла и уровень защиты паролем. Вы можете открыть файл в одном из трех режимов: Locked (заблокированный), Readonly (только для чтения), или Shared (совместно используемый). Режим Locked является режимом по умолчанию и означает, что никакой другой пользователь не может получить доступ к файлу, пока он открыт. Режим Readonly означает, что другие пользователи могут получить доступ к файлу, но только прочитать его.

Режим Shared означает, что другие пользователи могут одновременно открыть и получить доступ к файлу. Вы можете использовать встроенные функции LOCK и UNLOCK для блокирования и разблокирования отдельных записей в файле, чтобы исключить коллизии с другими пользователями.

Если вы присваиваете файлу пароль, вы можете также присвоить уровень защиты, который обеспечивает пароль. Уровни защиты следующие: Read, Write и Delete. Read означает, что вы должны ввести пароль, чтобы прочитать файл. Write означает, что вы можете считать файл, но должны предоставить пароль для записи в файл. Delete означает, что вы можете считать файл или записать в него, но вы не можете удалить файл без пароля.

**Примечание**: защита файла паролем и блокирование/разблокирование отдельных записей доступны не во всех реализациях. См. *Приложение 1 в Справочном руководстве языка PL/I*.

Атрибут LINESIZE применяется только к потоковым (STREAM) файлам и определяет максимальное количество символов в длине строки ввода или вывода. Атрибут PAGESIZE применяется только к потоковым выходным (STREAM OUTPUT) файлам и определяет количество строк на странице при выводе.

### Атрибуты файлов

PL/I контролирует все файловые операции через внутреннюю структуру данных, называемую блоком параметров файла (FPB[[3]](#footnote-3)). FPB содержит информацию о файле, в том числе, открыт он или закрыт, внешнее устройство или файл, связь с файлом, текущем номере строки и позиции или записи, к которой осуществляется доступ, и размере внутреннего буфера. Также FPB содержит дескриптор файла, который описывает атрибуты файла. Эти атрибуты, в свою очередь, описывают допустимые методы доступа. *Таблица 4-1* приводит допустимые атрибуты, которые можно назначить файлу, явно с помощью оператора OPEN, или неявно, осуществляя доступ с помощью оператора ввода-вывода.

Таблица 4‑1 Допустимые файловые атрибуты

|  |  |
| --- | --- |
| **Тип** | **Атрибут** |
| STREAM | INPUT ENVIRONMENT TITLE |
| STREAM | OUTPUT ENVIRONMENT TITLE LINESIZE |
| STREAM | PRINT ENVIRONMENT TITLE LINESIZE PAGESIZE |
| RECORD | INPUT SEQUENTIAL ENVIRONMENT TITLE |
| RECORD | OUTPUT SEQUENTIAL ENVIRONMENT TITLE |
| RECORD | INPUT SEQUENTIAL KEYED ENVIRONMENT TITLE |
| RECORD | OUTPUT SEQUENTIAL KEYED ENVIRONMENT TITLE |
| RECORD | INPUT DIRECT KEYED ENVIRONMENT TITLE |
| RECORD | OUTPUT DIRECT KEYED ENVIRONMENT TITLE |
| RECORD | UPDATE DIRECT KEYED ENVIRONMENT TITLE |

### Подразумеваемые атрибуты

Если вы не открываете файл с явными атрибутами, PL/I определяет атрибуты в зависимости от типа используемого оператора ввода-вывода для получения доступа к файлу. *Таблица 4-2* перечисляет подразумеваемые атрибуты, каждого из операторов ввода-вывода. В следующей таблице f - файловая константа, x - тип скалярных или агрегированных данных, который не является CHARACTER VARYING, и k - значение ключа FIXED BINARY.

Таблица 4‑2 Допустимые файловые атрибуты операторов ввода-вывода

| **Оператор ввода-вывода** | **Допустимые атрибуты** |
| --- | --- |
| GET FILE(f) LIST | STREAM INPUT |
| PUT FILE(f) LIST | STREAM OUTPUT |
| GET FILE(f) EDIT | STREAM INPUT |
| PUT FILE(f) EDIT | STREAM OUTPUT |
| READ FILE(f) INTO(x) | STREAM INPUT |
| READ FILE(f) INTO(x) | RECORD INPUT SEQUENTIAL |
| READ FILE(f) INTO(x) KEYTO(k) | RECORD INPUT SEQUENTIAL  KEYED ENVIRONMENT(Fixed(i)) |
| READ FILE(f) INTO(x) KEY(k) | RECORD INPUT DIRECT KEYED ENVIRONMENT(Fixed(i))  RECORD UPDATE DIRECT KEYED ENVIRONMENT(Fixed(i)) |
| WRITE FILE(f) FROM(v) | STREAM OUTPUT |
| WRITE FILE(f) FROM(x) | RECORD OUTPUT SEQUENTIAL |
| WRITE FILE(f) FROM(x) KEYFROM(k) | RECORD OUTPUT DIRECT KEYED ENVIRONMENT(Fixed(i))  RECORD UPDATE DIRECT KEYED ENVIRONMENT(Fixed(i)) |

### Закрытие файлов

Оператор CLOSE отсоединяет файл от внешнего набора данных. Оператор CLOSE имеет форму

**CLOSE FILE(file\_id);**

где file\_id - файловая константа, для которой PL/I очищает внутренние буферы, записывает все данные на диск и закрывает файл на уровне операционной системы. Вы можете впоследствии вновь открыть тот же файл, используя оператор OPEN. PL/I автоматически закрывает все открытые файлы в конце программы или после выполнения оператора STOP.

### Методы доступа к файлу

PL/I поддерживает два метода доступа к файлу:

* Потоковый ввод-вывод (STREAM)
* Записеориентированный ввод-вывод (RECORD)

Существуют три различных вида потокового ввода-вывода:

* *Управляемый списком*, использует операторы GET LIST и PUT LIST, которые передают список элементов данных без любых спецификаций формата.
* *Строчно-ориентированный*, использует операторы READ и WRITE, которые позволяют получить доступ к символьным данным переменной длины в неотредактированном виде. Эти операторы могут быть не доступны в других реализациях PL/I.
* *Управляемый редактированием*, использует операторы GET EDIT и PUT EDIT, которые предоставляют форматированный доступ к элементам символьных данных.

Ввод-вывод управляемый редактированием похож на ввод-вывод управляемый списком за исключением того, что он записывает данные в конкретные поля выходной строки, описанные в списке элементов формата. Список данных определяет количество записываемых значений в фиксированных полях, определенных списком формата.

Список формата может содержать два вида элементов формата: *элементы формата данных* и *элементы формата управления*. PL/I сопоставляет каждый элемент списка данных с элементом в списке формата. Элемент формата определяет, как PL/I интерпретирует элемент данных. Элементы формата управления PL/I выполняет по мере их появления в списке формата.

Любому элементу формата может предшествовать положительное значение целочисленной константы, не превышающее 254, которое определяет число повторных применений элемента формата или группы элементов формата.

### Элементы формата данных

Следующие примеры показывают различные элементы формата, которые вы можете использовать в операторе GET EDIT или PUT EDIT.

**A[(w)]**

Формат A читает или пишет следующее алфавитно-цифровое поле, ширина которого определена w, с усечением или дополнением пробелами справа. Если вы опускаете w, формат A использует в качестве ширины поля размер преобразованных символьных данных.

**B[nl[(w)]**

Формат B читает или записывает значения битовой строки. nl определяет число используемых битов для представления каждой цифры. w - ширина поля, которую вы должны использовать при вводе.

**E(w[,d])**

Формат E читает или пишет элемент данных в поле из w символов в экспоненциальном представлении с максимальной точностью, позволенной в ширине поля. w должен быть должен быть не менее 8.

**F(w[,d])**

Формат F читает или записывает значения с фиксированной точкой в поле шириной w цифр и d необязательных цифр справа от десятичной точки.

### Элементы управления формата

**LINE(ln)**

Перемещения к строке, определенной ln в потоке данных прежде, чем записать следующий элемент данных.

**COLUMN(nc)**

Перемещение к позиции столбца, определенного nc в потоке данных перед чтением или записью следующего элемента данных. Это может очистить текущую строку.

**PAGE**

Выполняет прогон страницы для файлов PRINT.

**SKIP[(nl))**

Пропуски nl строк прежде, чем прочитать или записать следующий элемент данных.

**X(n)**

Помещает n символов пробела в поток данных прежде, чем считать или записать следующий элемент данных.

**R(fmt)**

Определяет удаленный формат. Это означает, что формат определен в другом месте в операторе FORMAT.

### Предопределенные файлы

PL/I имеет две предопределенные файловые константы под названием SYSIN - клавиатура консоли, и SYSPRINT - вывод на экран консоли. Эти файлы не требуется объявлять, если вы не делаете на них прямую ссылку в операторе OPEN или ввода-вывода.

SYSIN по умолчанию имеет атрибуты:

**STREAM INPUT ENVIRONMENT(Locked,Buff(128)) TITLE('$CON') LINESIZE(80) PAGESIZE(0)**

SYSPRINT по умолчанию имеет атрибуты:

**STREAM PRINT ENVIRONMENT(Locked,Buff(128)) TITLE('$CON') LINESIZE(80) PAGESIZE(0)**

## Операторы обработки состояний

PL/I имеет ряд функций, которые делают его идеальным для программирования приложений. Одной из таких функций является возможность обработки состояний. В большинстве языков, программа не может восстановиться при возникновении состояния ошибок, таких как недопустимое преобразование данных, управление возвращается в операционную систему.

PL/I имеет различные функциональные возможности, которые позволяют перехватывать ошибки времени выполнения, программировать ответ и восстанавливать управление. Совокупность этих функциональных возможностей называется *обработкой состояния*.

PL/I обеспечивает обработку состояния с помощью следующих выполняемых операторов:

* ON
* REVERT
* SIGNAL

### Оператор ON

Оператор ON используется для перехвата и ответа программы на состояние, сообщенное системой во время выполнения, или выполнение оператора SIGNAL. Оператор ON - выполняемый оператор, который определяет ответ. Он имеет следующий формат:

**ON имя\_состояния ON-блок;**

где имя\_состояния - одна из основных категорий состояния, с или без подкода (см. [*Раздел 4.4.4*](#_Категории_состояний)). ON-тело является оператором или группа операторов PL/I, которые выполняются при возникновении состояния.

Если подкод отсутствует, то PL/I обрабатывает оператор ON, когда возникает состояние с любым из подкодов. Это эквивалентно подкоду 0. Файл состояния должен ссылаться на файл, для которого сообщается состояние.

### Оператор REVERT

Вы используете оператор REVERT, чтобы отключить ON состояние установленное оператором ON. Это важно, потому что у вас могут быть только шестнадцать установленных ON состояний, не переполняя область кода состояний. Если переполнение происходит, система времени выполнения PL/I прекращает обработку. Форма оператора REVERT:

**REVERT имя\_состояния;**

PL/I автоматически отменяет ON состояние установленное в данном блоке, когда управление покидает среду этого блока.

### Оператор SIGNAL

Оператор SIGNAL позволяет вам активировать ответ на состояние. Форма оператора SIGNAL следующая:

**SIGNAL имя\_состояния;**

### Категории состояний

Категории состояний описывают различные состояния, которые может устанавливать система времени выполнения или ваша программа, выполняя оператор SIGNAL.

Имеется девять основных категорий состояний с подкодами, некоторые из которых определены системой, и некоторые из которых Вы можете определить сами. *Таблица 4-3* показывает предопределенные подкоды.

Таблица 4‑3 Категории состояний PL/I и подкоды

| **Тип** | **Значение** |
| --- | --- |
| **ERROR** |  |
| ERROR(0) | Любой подкод ERROR |
| ERROR(1) | Преобразование данных |
| ERROR(2) | Переполнение стека ввода-вывода |
| ERROR(3) | Недопустимый аргумент функции |
| ERROR(4) | Конфликт ввода-вывода |
| ERROR(5) | Переполнение стека формата |
| ERROR(6) | Недопустимый элемент формата |
| ERROR(7) | Исчерпание свободной памяти |
| ERROR(8) | Ошибка оверлея, нет файла |
| ERROR(9) | Ошибка оверлея, недопустимый диск |
| ERROR(10) | Ошибка оверлея, размер |
| ERROR(11) | Ошибка оверлея, размещение |
| ERROR(12) | Ошибка оверлея, ошибка чтения диска |
| ERROR(13) | Недопустимый вызов ОС |
| ERROR(14) | Неудачная запись |
| ERROR(15) | Файл не открыт |
| ERROR(16) | Файл не с ключом |
| **FIXEDOVERFLOW** |  |
| FIXEDOVERFLOW(0) | Любой подкод FIXEDOVERFLOW |
| **OVERFLOW** |  |
| OVERFLOW(0) | Любой подкод OVERFLOW |
| OVERFLOW(1) | Операция с плавающей точкой |
| OVERFLOW(2) | Преобразование точности с плавающей точкой |
| **UNDERFLOW** |  |
| UNDERFLOW(0) | Любой подкод UNDERFLOW |
| UNDERFLOW(1) | Операция с плавающей точкой |
| UNDERFLOW(2) | Преобразование точности с плавающей точкой |
| **ZERODIVIDE** |  |
| ZERODIVIDE(0) | Любой подкод ZERODIVIDE |
| ZERODIVIDE(1) | Десятичное деление |
| ZERODIVIDE(2) | Деление с плавающей точкой |
| ZERODIVIDE(3) | Целочисленное деление |
| **ENDFILE** |  |
| **UNDEFINEDFILE** |  |
| **KEY** |  |
| **ENDPAGE** |  |

Кроме этих стандартных системных подкодов состояний можно определить некоторые подкоды в конкретном приложении для проверки нужного состояния и затем использовать оператор SIGNAL для обозначения состояния.

### Встроенные функции обработки состояний

PL/I предоставляет некоторые встроенные функции для помощи в обработке возникающих состояний. Этими функциями являются:

|  |  |
| --- | --- |
| * ONCODE * ONFILE * ONKEY | * PAGENO * LINENO |

Функция ONCODE возвращает подкод последний раз сообщенного состояния или ноль, если состояние не было сообщено.

Функция ONFILE возвращает внутреннее имя файла, участвующего в операции ввода-вывода, которая привела к возникновению состояния.

Функция ONKEY возвращает значение последнего ключа, участвующего в операцию ввода-вывода, которая привела к возникновению состояния.

Функции PAGENO и LINENO возвращают текущий номер страницы и номер строки для файла PRINT, названного в параметре.

## Операторы управления памятью

Каждая переменная в программе PL/I имеет атрибут класса памяти. Класс памяти определяет, как и когда PL/I выделяет хранение для переменной, и имеет ли переменная свое собственное хранение или совместно использует хранение с другой переменной.

PL/I поддерживает четыре различных класса памяти

* AUTOMATIC (по умолчанию в PL/I)
* BASED
* PARAMETER
* STATIC

PL/I обрабатывает память AUTOMATIC как память STATIC, кроме процедур, отмеченных как RECURSIVE. Компилятор выделяет память для статических переменных (STATIC) до выполнения, и она остается выделенной, пока программа работает. Вы можете использовать атрибут INITIAL для присвоения начальные постоянных значений статическим элементам данных.

Если переменная появляется в списке параметров, компилятор присваивает ей класс памяти PARAMETER. Область памяти для параметров выделяется, когда процедура вызова передает параметры в вызванную процедуру. (См. *Приложение 1 в Справочном руководстве языка PL/I*.)

**Примечание**: Только статические переменные могут иметь атрибут INITIAL, для совместимости со стандартом ANSI PL/I подмножества G.

Атрибуты класса памяти являются свойствами скаляров, массивов, переменных основных структур и файловых переменных. Вы не можете присвоить атрибуты класса памяти именам точек входа (entry), файловым константам или элементам агрегатов данных.

### Переменные BASED и указатели

Компилятор не выделяет память для переменных с классом памяти BASED. *Базированная переменная* - переменная, которая описывает область памяти, доступ к которой можно получить с помощью указателя. *Указатель* - адрес, где начинается область памяти базированной переменной, и сама базированная переменная определяет, как PL/I интерпретирует содержимое области памяти, расположенной в этом месте. Таким образом, базированная переменная вместе с указателем эквивалентна небазированной (обычной) переменной.

Можно представить себе базированную переменную как шаблон, который накладывается на область памяти, определенный его базой. Таким образом, базированная переменная может ссылаться на область памяти, выделенную для самой базированной переменной, или на область памяти, выделенную для других переменных.

Объявление переменной типа BASED имеет вид:

**DECLARE имя BASED [(указатель)];**

Например,

**declare A(5,5) character(10) based;  
declare bit\_vector bit(8) based(p);**

где указатель - неиндексированная переменная типа POINTER или вызов функции без параметров, которая возвращает значение POINTER.

Определенная с помощью указателя ссылка может быть неявной или явной. Если переменная объявлена как BASED без указателя, то каждая ссылка на переменную в программе должна включать явный спецификатор указателя следующего вида:

**выражение\_указателя -> переменная**

Если переменная объявлена, как BASED с указателем, можно ссылаться на нее без спецификатора указателя. Система времени выполнения повторно вычисляет ссылку указателя при каждом возникновении неполной переменной, используя выражение указателя, заданное в объявлении переменной.

Следующая последовательность кода иллюстрирует понятие базированных переменных.

**declare  
 p pointer,  
 a character(128),  
 b(128) character(1) based(p),  
 c(0:127) bit(8) based(p),  
 d(64) bit(16) based(p),  
 e(8,0:15) bit(8) based(p);  
 ...  
p = addr(a);  
 ...**

В этом примере, после того, как указатель p присваивается адрес a, каждая из переменных b, c, d и e относится к тем же 128 байтам хранения, занятого переменной a, несмотря на то, что они делают это по-разному. Таким образом переменные b, c, d и e накладываются на переменную a.

Существует один важный момент, который нужно учитывать. Наложения, приведенные выше, зависят от способа представления и хранения элементов данных конкретным процессором. Такой код делает программу зависящей от реализации. Например, в реализациях отличных от PL/I, внутреннее представление массива может включать некоторые байты заголовка в дополнение к байтам, используемым для представления элементов данных. В каждом случае вы должны исследовать внутреннее представление перед использованием базированных переменных для наложения других типов данных.

### Оператор ALLOCATE

Оператор ALLOCATE явно выделяет память для базированной переменной. Оператор ALLOCATE имеет форму:

**ALLOCATE базированная\_переменная SET(переменная\_указатель);**

Например,

**allocate input\_buffer set(buffer\_ptr);**

Система времени выполнения получает достаточную память для базируемой переменной из свободной области памяти и затем присваивает переменной указателя адрес этого сегмента памяти.

### Оператор FREE

Оператор FREE освобождает память, выделенную базированной переменной. Оператор FREE имеет форму:

**FREE [переменная\_указатель ->] базированная\_переменная;**

Например,

**free inputbuffer;**

**Примечание**: ссылка переменной указателя необязательная, если вы объявили базированную переменную со ссылкой указателя.

Следующая последовательность кода иллюстрирует использование операторов ALLOCATE и FREE.

**declare**

**(p,q,r) pointer,**

**a character based,**

**b fixed based(r);**

**...**

**allocate a set(p);**

**allocate b set(r);**

**allocate a set(q)**

**...**

**free p -> a;**

**free q -> a;**

**free b;**

**...**

## Операторы препроцессора

Операторы препроцессора позволяют вам включать другие файлы и изменять исходную программу во время компиляции.

Оператор %INCLUDE копирует источник PL/I из другого файла во время компиляции. Оператор %INCLUDE полезен для заполнения объявлений, которые повторяются на протяжении всей программы. Оператор %INCLUDE имеет форму:

**%INCLUDE 'имя\_файла';**

Например,

**%include 'fcb.dcl';**

Оператор %REPLACE позволяет во время компиляции заменять идентификаторы литеральными константами всюду по тексту программы PL/I. Оператор %REPLACE имеет форму:

**%REPLACE идентификатор BY константа;**

Вы можете поместить больше чем одну пару идентификатор-константа в один оператор %REPLACE, разделяя пары запятыми.

Например,

**%replace  
 true by '1'b,  
 false by '0'b;**

## Пустой оператор

Пустой оператор не выполняет действий. Он имеет простую форму:

;

Вы можете использовать пустой оператор в качестве цели THEN или ELSE в операторе IF. В следующем примере

**if x > average then  
 goto print\_it;  
else ;**

не выполняется ни каких действий, когда x меньше или равен average, выполнение продолжается с оператора, следующего за ELSE. В качестве другого примера, рассмотрим этот оператор:

**on endpage(report\_file);**

Здесь, когда PL/I обрабатывает ON-блок для ENDPAGE, ни какие действия не выполняются и оператор ввода-вывода, вызвавший состояние продолжает выполняться.

Вы можете также использовать пустые операторы, чтобы задать больше чем одну метку одному и тому же выполняемому оператору. Например,

**A: ;  
B: statement-1;  
 statement-2;**

**...**

Ссылки: Справочное руководство разделы с 2.7 по 2.9, 2.15, 7, 8, 9, с 10.1 по 10.3, с 10.7 по 10.8, 11.1, 11.3

**Конец раздела 4**

# Стиль программирования

Язык PL/I имеет свободный формат. Вы можете писать программы без учета позиций столбцов и конкретных форматов строк. Каждая строка может иметь длину до 120 символов, завершаемых возвратом каретки, и логически соединена со следующей строкой в последовательности. Компилятор просто читает исходную программу с первой до последней строки, игнорируя границы строки.

В обмен на эту свободу самовыражения вы должны придерживаться некоторых стилистических соглашений, чтобы ваши программы были легко читаемыми и понятны другим программистам. Профессиональная программа не только выдает правильный результат, но согласуется по форме и разделена на логические сегменты, которые легко понять. Логически структурированную программу также намного легче отлаживать. Хорошо созданная программа ценится за ее форму и ее функции.

Существует множество стилистических соглашений используемых отдельными программистами. Следующие правила описывают один из наборов соглашений, которые используются во всех примерах этого руководства. *Листинг 5-1* иллюстрирует соглашения, описанные в этом разделе.

## Регистр символов

Вы можете писать программы PL/I в верхнем или в нижний регистре. Внутренне, компилятор PL/I переводит все символы, за исключением строк в кавычках в верхний регистр. Использование нижнего регистра во всей программе в целом улучшает читаемость.

## Отступы

Используйте отступы по всей программе для выделения различных объявлений и операторов. Для упрощения добавления отступа, компилятор разворачивает табуляции (символы Ctrl-I) к каждой четвертой позиции столбца. Некоторые текстовые редакторы разворачивают символы табуляции до восьмой позиции, таким образом, строка кажется более широкой во время операции отображения и редактирования. Компилятор выдает ошибку TRUNC (усечение), если длина развернутой строки превышает 120 столбцов.

Операторы программы начинаются на внешнем уровне блока в первой позиции столбца. последующий уровень блока, инициируемый группой DO, блоками BEGIN или PROCEDURE, начинается на новом уровне отступа из четырех пробелов или одного символа табуляции. Вводите операторы в группе с тем же уровнем отступа с именами процедур и метками на одной строке.

Непосредственно за оператором IF должно следовать условие и ключевое слово THEN с последующим оператором, расположенным с отступом на следующей строке. Когда оператор IF имеет соответствующий ELSE, начинайте ELSE на том же уровне, что и IF. Расположите следующий оператор с отступом после ELSE и поместите его в следующей строке. Для объявлений, помещайте ключевое слово DECLARE на одной строке, а затем заявленные элементы с отступом на следующей строке.

Избегите сложных вынесений атрибутов за скобки, потому что это уменьшает удобочитаемость программы. Вставляйте пустые строки при необходимости, для улучшения структурирования и разделения логически различных сегментов программы.

## Сокращения

Многие из длинных ключевых слов PL/I имеют сокращения (см. *Список команд PL/I* для уточнения списка). Непоследовательное использование сокращений снижает читаемость, так что используйте либо длинные либо короткие формы, но не обе. Используйте подчеркивание в именах переменных для улучшения читабельности.

## Модульный формат

Вы должны разделить большие программы на несколько логических групп или модулей, где каждый модуль выполняет определенную примитивную функцию. Вы должны сделать эти модули подпрограммами PL/I, определенными локально или внешне. Локальные подпрограммы становятся частью той же главной программы или подпрограммы, в то время, как внешние подпрограммы вы можете компилировать отдельно и скомпоновать воедино.

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**/\* Эта программа вычисляет наибольшее из трех \*/**

**/\* чисел FLOAT BINARY x, y и z. \*/**

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**┌── test:**

**│ procedure options(main);**

**│ declare**

**│ (a,b,c) float binary;**

**│ put list ('Type Three Numbers: ');**

**│ get list (a,b,c);**

**│ put list ('The Largest Value is',max3(a,b,c));**

**│**

**│ /\* эта процедура вычисляет наибольшее из x, y и z \*/**

**│ ┌── max3: procedure(x,y,z) returns(float binary);**

**│ │ declare**

**│ │ (x,y,z,max) float binary;**

**│ │**

**│ │ ┌── if x > y then**

**│ │ │ ┌── if x > z then**

**│ │ │ │ max = x;**

**│ │ │ └── else**

**│ │ │ max = z;**

**│ │ └── else**

**│ │ ┌── if y > z then**

**│ │ │ max = y;**

**│ │ └── else**

**│ │ max = z;**

**│ │ return(max);**

**│ └── end max3;**

**└── end test;**

Листинг 5‑1 Стилистические соглашения PL/I

Поместите локально определенные подпрограммы в конце программы, чтобы в начале располагались только объявления и операторы верхнего уровня, которые вызывают локальные подпрограммы. Ни операторы верхнего уровня, ни локально определенные подпрограммы не должен превышать одну или две страницы в длину.

Во время обучения программированию на языке PL/I, используйте главную программу с локально определенными подпрограммами, следуя форме примеров, приведенных в данном руководстве. Однако, когда ваши прикладные программы увеличатся в размере, вы сочтете более эффективным разделить их на отдельные модули. Это позволяет компилировать и компоновать отдельные сегменты по частям, тем самым уменьшая общее время разработки.

## Комментарии

Комментарии должны стать неотъемлемой частью вашей программы. Они являются существенным элементом в создании вами программы легко читаемой для себя и другими программистами. Избегайте вставки случайных комментариев по всему исходному файлу и не вкладывайте их. Системность является лозунгом. Поместите свои комментарии в начале подпрограмм или логических групп операторов. Если ваша программа правильно структурирована в четко определенные модули, эти поясняющие комментарии предоставляют информацию, необходимую для понимания общего предназначения и работы программы. Они также упрощают задачу поддержки и обновления кода без внесения ошибок.

Ссылки: Список команд PL/I

**Конец раздела 5**

# Использование системы

Разработка программы PL/I представляет собой процесс из 3 этапов:

1. Написание исходного файла, используя любой подходящий текстовый редактор.
2. Компиляция исходного файла и генерация перемещаемого объектного файла.
3. Соединение перемещаемого объектного файла с библиотекой подпрограмм времени выполнения для генерации выполняемого командного файла.

PL/I является компилируемым языком. Следовательно, если вы вносите какое-либо изменение в исходный файл, вы должны перекомпилировать программу. Попытайтесь разделить большие программы на несколько маленьких модулей, скомпилируйте каждый модуль отдельно, затем соединяйте их. Небольшие программы компилируют быстрее и используют меньше памяти для таблицы символов.

*Рисунок 6-1* иллюстрирует процесс разработки.



Рисунок 6‑1 Разработка программы PL/I

## Системные файлы PL/I

Когда вы получаете свою систему PL/I, вы должны сначала сделать копии всех дистрибутивных дисков. Если вы не знаете, как это сделать, читайте в документации по операционной системе.

Содержание дистрибутивных дисков меняется в зависимости от версии. Файл RELNOTES.PRN на диске с примерами программ описывает содержание файлов для вашей конкретной версии.

**Примечание**: у вас есть определенные обязанности при создании копий Digital Research. Убедитесь, что вы прочитали ваше лицензионное соглашение.

После того, как вы сделаете диски с резервной копией, загрузите ваш диск компилятора и введите команду DIR:

A>**dir**

Каталог содержит несколько типов файлов, приведенных в *Таблице 6-1*.

Таблица 6‑1 Системные файлы PL/I

|  |  |
| --- | --- |
| **Тип** | **Описание** |
| **CMD** | Выполняемый командный файл (версия 8086), например, DEMO.CMD |
| **COM** | Выполняемый командный файл (версия 8080), например, DEMO.COM |
| **DAT** | Тип файла данных по умолчанию |
| **DCL** | Файл %INCLUDE (объявления данных) |
| **EXE** | Выполняемый командный файл (в IBM DOS), например, DEMO.EXE |
| **IRL** | Индексируемый перемещаемый файл, например PLILIB.IRL |
| **L86** | Файл библиотеки (версия 8086), например, PLILIB.L86 |
| **OBJ** | Файл перемещаемого объектного кода (версия 8086), например, DEMO.OBJ |
| **OVL** | Оверлейные программы компилятора PL/I (версия 8080), PLI0, PLI1 и PLI2 |
| **OVR** | Оверлейные программы компилятора PL/I (версия 8086), PLI0, PLI1 и PLI2 |
| **PLI** | Исходные программы PL/I, например, DEMO.PLI |
| **PRN** | Дисковый файл принтера. Распечатка скомпилированной программы на диске. Также используется для читаемого файла документации. |
| **REL** | Файл перемещаемого объектного кода (версия 8080), например, DEMO.REL |
| **SYM** | Файл таблицы символов, например DEMO.SYM |

**Примечание**: файлами, которые содержат печатаемые символы являются только исходные программы PLI, файлы распечатки для принтера PRN и файлы таблиц символов SYM.

## Вызов компилятора

Вы вызываете компилятор PL/I, используя команду общего вида:

**pli спецификация\_файла [$опции]**

где спецификация файла определяет компилируемую программу и может включать дополнительную спецификацию диска. Например,

**d:myfile.pli**

Вы не должны определять тип файла, потому что компилятор принимает тип PLI.

$опции представляют список параметров, которые вы можете дополнительно включать в командную строку при компиляции программы. Эти параметры включают различные опции компилятора, приведенные в *Таблице 6-2*.

В каждом случае однобуквенная опция следует за символом $ в командной строке. Вы можете определить максимум до семи опций после знака доллара. Режим по умолчанию, не используя опций, компилирует программу, но не производит исходный листинг и отправляет все сообщения об ошибках на консоль.

Таблица 6‑2 Опции компилятора PL/I

|  |  |
| --- | --- |
| **Опция** | **Включенное действие** |
| **A** | *Сокращенный листинг*. Отключает листинг параметров и листинг операторов %INCLUDE во время первого прохода компилятора. |
| **B** | *Встроенная трассировка подпрограммы*. Во время выполнения показывает библиотечные функции, которые вызываются вашей программой PL/I. |
| **D** | *Печать файла на диск*. Отправляет листинг файла на диск, используя тип файла PRN. |
| **I** | *Чередование исходного и машинного кода*. Декодирует машинный код, произведенный компилятором в форме псевдо-ассемблера. |
| **K** | Аналогична A (реализации 8080) |
| **L** | *Листинг исходной программы*. Производит листинг исходной программы с номерами строк и размещением машинного кода (автоматически устанавливается переключателем I). |
| **N** | *Отображение уровня вложенности*. Включает на 1 проходе трассировку, которая показывает точный баланс операторов DO, PROCEDURE и BEGIN с соответствующими им операторами END. |
| **O** | *Отмена объектного кода*. Отключает вывод перемещаемого объектного кода, обычно создаваемого компилятором. |
| **P** | *Режим печати страниц*. Вставляет символы прогона страницы через каждые 60 строк и отправляет листинг на принтер. |
| **S** | *Отображение таблицы символов*. Показывает имена переменных программы, вместе с присвоенными им, назначенными по умолчанию и расширенными атрибутами. |

## Работа компилятора

Компилятор PL/I читает исходные программные файлы и генерирует перемещаемый объектный файл с машинно-зависимым кодом, в качестве выхода. PL/I является 3-х проходным компилятором с каждым проходом в виде отдельного оверлея. Проход 1 собирает объявления и создает таблицу символов, используемую последующими проходами. Проход 2 обрабатывает выполняемые операторы, увеличивает таблицу символов и генерирует промежуточный язык в форме древовидной структуры. Оба прохода анализируют исходный текст, используя метод рекурсивного спуска.

Проход 3 выполняет генерацию абсолютного кода, и включает в себя полный оптимизатор кода, который обрабатывает промежуточные древовидные структуры. Альтернативные формы эквивалентного выражения сводятся к одной и той же форме, и выражения переставляются для уменьшения количество временных переменных. Существует также устройство распознавания специальных форм, которое обнаруживает и подбирает приблизительно триста особо значимых древовидных структур. Распознавание специальных форм позволяет компилятору генерировать компактную последовательность кода для многих общих операторов.

**Примечание**: Все оверлейные программы компилятора (PLI0, PLI1 и PLI2) должны быть на диске по умолчанию.

По мере выполнения компилятором первых двух проходов, он выводит на экран сообщения:

**NO ERROR(S) IN PASS 1  
NO ERROR(S) IN PASS 2**

Если есть ошибки, компилятор выводит каждую строку, содержащую ошибку с номером строки слева, короткое сообщение об ошибке и ниже символ ? в позиции строки, где возникает ошибка.

В конце 3-го прохода, компилятор выводит на экран сообщение:

**CODE SIZE = nnnn  
DATA AREA = nnnn  
FREE SYMS = nnnn  
END COMPILATION**

где nnnn - шестнадцатеричные числа, представляющие объем памяти, используемой кодом и данными, а также объем области транзитных программ (TPA[[4]](#footnote-4)) используемой для таблицы символов.

Если число сообщений об ошибках является чрезмерным, и вы хотите внести исправления, прежде чем продолжить, вы можете прервать компиляцию, нажав любую клавишу. Компилятор отвечает с сообщением:

**STOP PL/I (Y/N)?  
Enter Y to halt the compilation.**

**Примечание**: В DOS, вы останавливаете программу при нажатии Ctrl-Break, которое сразу возвращает управление операционной системе. Поэтому вы не будете видеть сообщение, выведенное PL/I.

Если вы используете опцию N, компилятор выводит слева номер строки программы, за которой следует буквы от a до z, обозначающие уровень вложенности для каждой строки. Основной уровень программы обозначается a, и каждый вложенный BEGIN, повышает уровень на одну букву, а каждая вложенная PROCEDURE повышает уровень на две буквы.

Если вы используете опцию L, компилятор выводит относительный адрес машинного кода для каждой строки в виде четырехзначного шестнадцатеричного числа. Этот адрес используется для определения количества машинного кода, сгенерированного для каждого оператора и относительного адреса машинного кода для каждой строки программы. Компилятор распечатывает оператор исходного языка на следующей строке после относительного значения машинного кода.

*Листинги 6-1* и *6-2* показывают две компиляции программы под названием DEMO, которая находится на вашем диске с примерами программ.

**1 a ┌── demo:**

**2 b │ procedure options(main);**

**3 b │**

**4 b │ declare**

**5 b │ name character(20) varying;**

**6 b │**

**7 b │**

**8 b │ put skip(2) list('PLEASE ENTER YOUR FIRST NAME: ');**

**9 b │ get list(name);**

**10 b │ put skip(2) list('HELLO '||name||', WELCOME TO PL/I');**

**11 b │**

**12 b └── end demo;**

Листинг 6‑1 Компиляция DEMO с опцией $N

**1 a 0000┌── demo:**

**2 a 0006│ procedure options(main);**

**3 a 0006│**

**4 c 0006│ declare**

**5 c 0006│ name character(20) varying;**

**6 c 0006│**

**7 c 0006│**

**8 c 0006│ put skip(2) list('PLEASE ENTER YOUR FIRST NAME: ');**

**9 c 0022│ get list(name);**

**10 c 003C│ put skip(2) list('HELLO '||name||', WELCOME TO PL/I');**

**11 c 006A│**

**12 a 006A└── end demo;**

Листинг 6‑2 Компиляция DEMO с опцией $L

## Программа DEMO

Вы можете начать учиться использовать систему PL/I, компилируя программу под названием DEMO. Исходный файл для DEMO находится на вашем диске с примерами программы PL/I, таким образом, вы не должны писать код. Чтобы вывести на экран исходный файл, используйте команду TYPE, следующим образом:

A>**type demo.pli**

Чтобы скомпилировать программу DEMO, введите команду:

A>**pli demo**

Теперь исследуйте свой каталог и найдите объектный файл, который содержит перемещаемый машинный код, произведенный компилятором. Машинный код, произведенный компилятором, не является непосредственно исполнимой программой, таким образом, вы должны соединить объектный файл с библиотекой подпрограмм времени выполнения с помощью команды:

A>**link demo**

Теперь исследуйте свой каталог и найдите командный файл и файл таблицы символов произведенные редактором связей. Вы можете загрузить файл таблицы символов под управлением SID-80™ или SID-86™ для отладки.

## Выполнение DEMO

Чтобы выполнить скомпилированную программу, введите имя командного файла:

A>**demo**

Операционная система загружает программу DEMO, которая начинает выполнение и выводит сообщение:

PLEASE ENTER YOUR FIRST NAME:

Консольный ввод представляет собой поле произвольного размера и обеспечивает все возможности редактирования строки операционной системы. Когда вы вводите свое имя, DEMO дает адекватный ответ. *Листинг 6-3* показывает диалог с DEMO.

A>**demo**PLEASE ENTER YOUR FIRST NAME: **Larry**  
HELLO Larry, WELCOME TO PL/I  
A>

Листинг 6‑3 Диалог с программой DEMO

Различные ошибки времени выполнения могут остановить обработку, если программа явно не перехватывает их. В этом случае PL/I выводит на экран сообщение в следующей форме:

**ошибка\_состояния (код), опции\_файла, вспомогательное\_сообщение  
Traceback: aaaa bbbb cccc dddd # eeee ffff gggg hhhh**

Ошибка\_состояния соответствует одной из стандартных категорий состояний PL/I (см. *Раздел 4.4.4*). Код является подкодом ошибки, идентифицирующим источник ошибки.

PL/I распечатывает опции\_файла, когда ошибка подразумевает операцию ввода-вывода и принимает форму:

**File: внутреннее\_имя=внешнее\_имя**

где внутреннее\_имя это имя программы, которая ссылается на файл, участвующий в ошибке, и внешнее\_имя это внешнее устройство или имя файла, связанное с файлом. PL/I распечатывает вспомогательное\_сообщение каждый раз, когда предыдущая информация недостаточна для идентификации ошибки.

Часть traceback перечисляет до восьми элементов внутреннего стека. В предыдущей общей форме элемент aaaa соответствует вершине стека, в то время как hhhh соответствует нижней части стека. Если глубина стека превышает восемь элементов, символ # разделяет четыре верхние элемента слева от четырех нижних элементов справа.

*Листинг 6-4* содержит пример диагностической формы. В этом случае в консоли введен символ конца файла (^Z). Ввод ^Z приводит к состоянию ENDFILE для файла SYSIN. Это - стандартный консольный ввод. В этом примере внешним устройством, соединенным с файлом SYSIN, является консоль, обозначенная CON.

A>**demo**

PLEASE ENTER YOUR FIRST NAME: **^Z**

END OF FILE (1), File: SYSIN=CON

Traceback: 07BE 0769 012E 4COO # 0702 0322 8090 012E

A>

Листинг 6‑4 Обратная трассировка ошибки в программе DEMO

## Сообщения об ошибках и коды состояний

PL/I может обнаружить два вида ошибок: ошибки компиляции и ошибки периода выполнения. Компилятор отмечает каждую ошибку компиляции с помощью символа ? около позиции ошибки в строке и выводит сообщение об ошибке после строки, содержащей ошибку. Символ ? может отстоять от позиции фактической ошибки на несколько столбцов. В некоторых случаях ошибка в одной строке может привести к ошибкам в последующих строках.

PL/I подразделяет ошибки на восстанавливаемые и невосстанавливаемые. Большинство ошибок компиляции восстанавливаемые, и компилятор продолжает обрабатывать исходный файл. Однако некоторые ошибки компиляции невосстанавливаемые. Компилятор прекращает обработку, и управление сразу возвращается в операционную систему.

Система времени выполнения обнаруживает ошибки во время выполнения программы. Большинство ошибок периода выполнения восстанавливаемые, если перехвачены ON-блоком. Однако некоторые ошибки периода выполнения невосстанавливаемые. Программа завершается, и управление сразу возвращаются в операционную систему.

В большинстве случаев сообщения об ошибках зависят от конкретной реализации. См. полный перечень в *Приложении 5 Справочного руководства PL/I*.

**Конец раздела 6**

# Использование данных различных типов

Программы PL/I программы позволяют использовать различные типы данных для различных применений. В программах по всему руководству, вы должны обратить внимание каким образом и почему каждый тип данных используется в той или иной ситуации.

## Программа FLTPOLY

*Листинг 7-1* показывает программу для вычисления полиномиального выражения. Программа начинается, читая три значения: x, y и z с консоли, и затем использует значения для вычисления полиномиального выражения:

**p(x,y,z) = x2 + 2y + z**

Основная часть программы ограничена единственной DO-группой. На каждой следующей итерации, программа считывает значения x, y и z из стандартного файла SYSIN, консоли. Затем программа записывает значение, произведенное p(x,y,z) в файл SYSPRINT, снова, консольный файл. Наконец, если все входные значения равны нулю, программа выполняет оператор STOP и заканчивает бесконечный цикл.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет полиномиальное выражение, \*/**

**3 a /\* используя данные FLOAT BINARY. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── fltpoly:**

**6 b │ procedure options(main);**

**7 b │**

**8 b │ %replace**

**9 b │ true by '1'b;**

**10 b │ declare**

**11 b │ (x,y,z) float binary(24);**

**12 b │**

**13 c │ ┌── do while(true);**

**14 c │ │ put skip(2) list('Type x,y,z: ');**

**15 c │ │ get list(x,y,z);**

**16 c │ │**

**17 c │ │ if x=0 & y=0 & z=0 then**

**18 c │ │ stop;**

**19 c │ │**

**20 c │ │ put skip list(' 2');**

**21 c │ │ put skip list(' x + 2y + z =',P(x,y,z));**

**22 c │ └── end;**

**23 b │**

**24 b │ ┌── P:**

**25 c │ │ procedure (x,y,z) returns (float binary(24));**

**26 c │ │ declare**

**27 c │ │ (x,y,z) float binary;**

**28 c │ │ return (x \* x + 2 \* y + z);**

**29 c │ └── end P;**

**30 b │**

**31 b └── end fltpoly;**

Листинг 7‑1 Программа вычисления полинома (FLOAT BINARY)

Программа в строке 8 использует оператор %REPLACE для определения литерального значения true равным константе строки битов '1'b. При каждом появлении имени true компилятор заменяет его этим значением. Таким образом, компилятор интерпретирует DO-группу, начинающую в строке 13 как

**do while('1'b);**

**...**

**end;**

бесконечный цикл, пока не выполняется содержащийся внутри него оператор STOP. Использование оператора %REPLACE для определения констант может улучшить удобочитаемость ваших программ.

*Листинг 7-2* показывает консольное взаимодействие с программой FLTPOLY. Начальные значения для x, y и z: 1.4, 2.3 и 5.67, но на следующем цикле, ввод принимает форму:

**,4.5,,**

Эта форма изменяет значение только y. Таким образом, на этой итерации, значения x, y и z равны 1.4, 4.5, и 5.67. Третья входная строка изменяет y и z, в то время как четвертая строка изменяет только x.

A>**fltpoly**

Type x,y,z: **1.4, 2.3, 5.67**

2

x + 2y + z = 1.2230000E+01

Type x,y,z: **,4.5,,**

2

x + 2y + z = 1.6629999E+01

Type x,y,z: **,4.0,-3.7**

2

x + 2y + z = 6.2600000E+00

Type x,y,z: **2.3,,,**

2

x + 2y + z = 9.5900000E+00

Type x,y,x: **0,0,0**

A>

Листинг 7‑2 Взаимодействие с программой FLTPOLY

## Программа DECPOLY

*Листинг 7-3* показывает программу DECPOLY, которая является по существу той же программой, что и в *Листинге 7-1*. Различие между этими двумя программами, что FLTPOLY использует элементы FLOAT BINARY, в то время как DECPOLY использует элементы FIXED DECIMAL. Вычисления FLOAT BINARY выполняются значительно быстрее, чем их эквиваленты FIXED DECIMAL, но вычисления FLOAT BINARY одинарной точности включают ошибки усечения и производят ответ только с точностью около 7 десятичных разрядов.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет полиномиальное выражение, \*/**

**3 a /\* используя данные FIXED DECIMAL. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── decpoly:**

**6 b │ procedure options(main);**

**7 b │**

**8 b │ %replace**

**9 b │ true by '1'b;**

**10 b │ declare**

**11 b │ (x,y,z) fixed decimal(15,4);**

**12 b │**

**13 c │ ┌── do while(true);**

**14 c │ │ put skip(2) list('Type x,y,z: ');**

**15 c │ │ get list(x,y,z);**

**16 c │ │**

**17 c │ │ if x=0 & y=0 & z=0 then**

**18 c │ │ stop;**

**19 c │ │**

**20 c │ │ put skip list(' 2');**

**21 c │ │ put skip list(' x + 2y + z =',P(x,y,z));**

**22 c │ └── end;**

**23 b │**

**24 b │ ┌── P:**

**25 c │ │ procedure (x,y,z) returns (fixed decimal(15,4));**

**26 c │ │ declare**

**27 c │ │ (x,y,z) fixed decimal(15,4);**

**28 c │ │ return (x \* x + 2 \* y + z);**

**29 c │ └── end P;**

**30 b │**

**31 b └── end decpoly;**

Листинг 7‑3 Программа вычисления полинома (FIXED DECIMAL)

*Листинг 7-4* показывает консольное взаимодействие с программой DECPOLY. Начальные входные значения для x, y и z: 1.4, 2.3, и 5.67. Эти же значения используются для программы FLTPOLY, но обратите внимание на разницу в выводе. Вторая итерация изменяет значения y и z, и третья итерация изменяет все три значения.

A>**decpoly**

Type x,y,z: **1.4,2.3,5.67**

2

x + 2y + z = 12.2300

Type x,y,z: **,.0006,7**

2

x + 2y + z = 8.9612

Type x,y,z: **723.445,80.54,0**

2

x + 2y + z = 523533.7480

Type x,y,z: **0,0,0**

A>

Листинг 7‑4 Взаимодействие с программой DECPOLY

Поэкспериментируйте с этими двумя программами путем сравнения результатов при вводе одинаковых значений в каждой из них. Понимание как PL/I внутренне обрабатывает данные различных типов позволит вам выбрать правильный тип данных в соответствии с приложением.

Ссылки: Справочное руководство разделы 3.1, 11.1, Приложение 1

**Конец раздела 7**

# Обработка файлов STREAM и RECORD

Примеры программы в этом разделе иллюстрируют обработку потоковых (с атрибутом STREAM) и записеориентированных (с атрибутом RECORD) файлов, используя различные операторы ввода-вывода.

## Программа копирования файла

*Листинг 8-1* показывает программу общего назначения копирования файла. Программа сначала объявляет и открывает две файловые константы с именами input\_file и input\_file. Затем она начинает выполнять непрерывный цикл, в котором считывает данные из входного файла и копирует их в выходной файл.

Оба оператора OPEN объявляют потоковые (STREAM) файлы с внутренним буфером 8192 байт каждый. В первом операторе OPEN PL/I по умолчанию предоставляет атрибут INPUT (входной), в то время как второй оператор OPEN явно определяет файл в качестве выходного с помощью атрибута OUTPUT. Иначе он по умолчанию также получил атрибут INPUT.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа копирует один файл в другой, \*/**

**3 a /\* используя буферированный ввод-вывод. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── copy:**

**6 b │ procedure options(main);**

**7 b │ declare**

**8 b │ (input\_file,output\_file) file;**

**9 b │**

**10 b │ open file (input\_file) stream**

**11 b │ environment(b(8192)) title('$1.$1');**

**12 b │**

**13 b │ open file (output\_file) stream output**

**14 b │ environment(b(8192)) title('$2.$2');**

**15 b │ declare**

**16 b │ buff character(254) varying;**

**17 b │**

**18 c │ ┌── do while('1'b);**

**19 c │ │ read file (input\_file) into (buff);**

**20 c │ │ write file (output\_file) from (buff);**

**21 c │ └── end;**

**22 b └── end copy;**

Листинг 8‑1 Программа COPY (копирования файла)

Эта программа показывает особое использование операторов READ и WRITE для обработки потоковых файлов (STREAM). Оператор READ в строке 19 читает потоковый файл в символьную строку переменной длины buff. Он считывает каждую входную строку до и включая следующие символы перевода строки и возврата каретки в buff и устанавливает длину buff равным количеству считанных данных, включая символы перевода строки и возврата каретки. Оператор WRITE выполняет противоположное действие. Он отправляет данные в потоковый файл из buff. Выходной файл получает все символы от первой позиции до длины buff.

Программа завершается путем чтения входного файла, пока он не достигнет символа конца файла (Ctrl-Z). PL/I автоматически закрывает все открытые файлы и записывает внутренние буферы на диск, таким образом, сохраняя недавно созданный выходной файл.

В *Листинге 8-2* показан пример выполнения программы COPY, используя следующую командную строку:

A>**copy copy.pli $con**

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Эта программа копирует один файл в другой, \*/

/\* используя буферированный ввод-вывод. \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

copy:

procedure options(main);

declare

(input\_file,output\_file) file;

open file (input\_file) stream

environment(b(8192)) title('$1.$1');

open file (output\_file) stream output

environment(b(8192)) title('$2.$2');

declare

buff character(254) varying;

do while('1'b);

read file (input\_file) into (buff);

write file (output\_file) from (buff);

end;

end copy;

END OF FILE (3), File: INPUT\_FILE=COPY.PLI

Traceback: 0524 0488 0163

Листинг 8‑2 Взаимодействие с программой COPY

В этом случае входным файлом является оригинальный исходный файл COPY.PLI первоисточника, а выходным файлом - системная консоль. Таким образом программа просто распечатывает COPY.PLI на терминале.

Опция TITLE соединяет внутренние имена файлов с внешними устройствами и файлами. Командная строка состоит из двух частей: самой команды и хвоста команды, который может содержать два имени файлов.

┌─ Команда ┌─────────┬─────── Хвост команды

│ ↓ ↓

↓ **$1 $2**

┌────────┬────────────┬────────┐

**│ copy │ copy.pli │ $con │**

└────────┴────────────┴────────┘

Рисунок 8‑1 Имена файлов по умолчанию в хвосте команды

Оператор OPEN в строке 10 берет в хвосте команды первое имя по умолчанию, включая диск (обозначенный $1.$1) и присваивает его внутренней файловой константе с именем input\_file. Аналогично, второй оператор OPEN в строке 13 берет в хвосте команды второе имя по умолчанию, включая диск (обозначенный на $2.$2) и присваивает его внутренней файловой константе с именем output\_file.

Например, команда

A>**copy a:x.dat c:u.new**

копирует файл X.DAT с диска a в новый файл U.NEW на диск c. Входной файл должен существовать, но PL/I стирает выходной файл, если он существует и создает его снова.

## Файл имен и адресов

Две программы, приведенные в *Листингах 8-3* и *8-6*, управляют простым файлом имен и адресов. Программа CREATE производит потоковый файл (с атрибутом STREAM), содержащий индивидуальные имена и адреса, к которым впоследствии получает доступ программа RETRIEVE.

### Программа CREATE

Программа CREATE в *Листинге 8-3* содержит структуру данных, которая определяет имя, адрес, город, регион, почтовый индекс и формат телефонного номера. Эта структура данных не находится в исходном файле CREATE.PLI. Она содержится в отдельном файле с названием RECORD.DCL, и CREATE использует оператор %INCLUDE, чтобы считать и объединить этот файл с исходным файлом. Оба файла находятся на вашем диске с примерами программ. Символы '+' справа от номера исходной строки в листинге указывают, что код поступает из %INCLUDE файла. Фактические строки в исходной программе выглядят следующим образом:

**create:  
 procedure options(main);  
  
 %include 'record.dcl';**

Файл, определенный в операторе %INCLUDE может быть любым допустимым именем файла. Компилятор просто копирует файл в точку расположения оператора %INCLUDE, и затем продолжает компиляцию.

Оператор OPEN, расположенный в строке 30, не определяет атрибут PRINT. Это означает, что выходной файл находится в форме, пригодной для последующего ввода с помощью оператора GET LIST.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* This program creates a name and address file. The \*/**

**3 a /\* data structure for each record is in the %INCLUDE \*/**

**4 a /\* file RECORD.DCL. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── create:**

**7 b │ procedure options(main);**

**8 b │**

Листинг 8‑3 Программа CREATE

**9+b │ dcl**

**10+b │ 1 record,**

**11+b │ 2 name character(30) varying,**

**12+b │ 2 addr character(30) varying,**

**13+b │ 2 city character(20) varying,**

**14+b │ 2 state character(10) varying,**

**15+b │ 2 zip fixed decimal(6),**

**16+b │ 2 phone character(12) varying;**

**17 b │**

**18 b │ %replace**

**19 b │ true by '1'b,**

**20 b │ false by '0'b;**

**21 b │**

**22 b │ declare**

**23 b │ output file,**

**24 b │ filename character(14) varying,**

**25 b │ eofile bit(1) static initial(false);**

**26 b │**

**27 b │ put list ('Name and Address Creation Program, File Name: ');**

**28 b │ get list (filename);**

**29 b │**

**30 b │ open file(output) stream output title(filename);**

**31 b │**

**32 c │ ┌── do while (^eofile);**

**33 c │ │ put skip(3) list('Name: ');**

**34 c │ │ get list(name);**

**35 c │ │ eofile = (name = 'EOF');**

**36 c │ │ if ^eofile then**

**37 d │ │ ┌── do;**

**38 d │ │ │ /\* write prompt strings to console \*/**

**39 d │ │ │ put list('Address: ');**

**40 d │ │ │ get list(addr);**

**41 d │ │ │ put list('City, State, Zip: ');**

**42 d │ │ │ get list(city, state, zip);**

**43 d │ │ │ put list('Phone: ');**

**44 d │ │ │ get list(phone);**

**45 d │ │ │**

**46 d │ │ │ /\* data in memory, write to output file \*/**

**47 d │ │ │ put file(output)**

**48 d │ │ │ list(name,addr,city,state,zip,phone);**

**49 d │ │ │ put file(output) skip;**

**50 d │ │ └── end;**

**51 c │ └── end;**

**52 b │ put file(output) skip list('EOF');**

**53 b │ put file(output) skip;**

**54 b │**

**55 b └── end create;**

Листинг 8-3 Программа CREATE (продолжение)

*Листинг 8-4* показывает консольное взаимодействие с программой CREATE. Вы определяете выходной файл как names.dat в первой входной строке. Оператор GET LIST, строка 34, принимает ввод, разграниченный пробелами и запятыми, если разделители не включены в одиночные апострофы. Таким образом CREATE, принимает входную строку

**'John Robinson**

как единственное строковое значение для которого PL/I автоматически вставляет подразумеваемый заключительный апостроф в конце строки. Последняя запись включает в себя три входных значения

**Unknown, 'Can"t Find', 99999**

которые CREATE, присваивает переменным city, street и state. Поскольку первое значение не начинается с апострофа, система ввода-вывода сканирует элемент данных до появления следующего пробела, символа табуляции, запятой, или конца строки. Второй элемент данных начинается с апострофа, и это заставляет систему ввода-вывода получать весь ввод до завершающего сбалансированного апострофа и уменьшать все встроенные двойные апострофы до единственного апострофа. Последнее значение, 99999, присваивается десятичному числу и должно содержать только числовые данные. Вы можете использовать команду

A>**type names.dat**

для отображения потокового файла, созданного программой. *Листинг 8-5* показывает результирующий вывод из каждой входной записи.

A>**create**

Name and Address Creation Program, File Name: **names.dat**

Name: **'Arthur Jackson'**

Address: **'100 W. 3rd St.'**

City, State, Zip: **'Fresno', 'Ca.', 93706**

Phone: **'529-1277'**

Name: **'Donna Harris'**

Address: **'12999 Sierra Rd.'**

City, State, Zip: **'Chico', 'Ca.', 95926**

Phone: **'635-3570'**

Name: **'John Robinson**

Address: **'1805 Franklin St.'**

City, State, Zip: **'Monterey', 'Ca.', 93940**

Phone: **'649-1000'**

Name: **'Virginia Wilson'**

Address: **'?'**

City, State, Zip: **Unknown, 'Can"t Find', 99999**

Phone: **'?'**

Name: **'EOF'**

A>

Листинг 8‑4 Взаимодействие с программой CREATE

A>**type names.dat**

'Arthur Jackson' '100 W. 3rd St.' 'Fresno' 'Ca.' 93706 '529-1277'

'Donna Harris' '12999 Sierra Rd.' 'Chico' 'Ca.' 95926 '635-3570'

'John Robinson' '1805 Franklin St.' 'Monterey' 'Ca.' 93940 '649-1000'

'Virginia Wilson' '?' 'Unknown' 'Can"t Find' 99999 '?'

'EOF'

A>

Листинг 8‑5 Вывод программы CREATE

### Программа RETRIEVE

Программа RETRIEVE, показанная в *Листинге 8-6*, читает файл, создаваемый CREATE, и выводит на экран данные имени и адреса по запросу пользователя. Компилятор включает тот же файл RECORD.DCL, используемый в программе CREATE, показанной в *Листинге 8-3*.

Основная DO-группа в программе RETRIEVE, между строками 31 и 60, читает два строковых значения, соответствующие самым низким (lower) и самым высоким (upper) именам, чтобы распечатать на каждой итерации. Встроенная DO-группа между строками 42 и 58 считывает весь входной файл и перечисляет только имена между нижними и верхними границами.

Программа RETRIEVE, подобно программе CREATE, читает имя исходного файла с консоли. Однако, RETRIEVE открывает и закрывает этот исходный файл каждый раз, когда получает запрос извлечения с консоли.

Оператор OPEN в строке 39 устанавливает размер внутреннего буфера входного файла равным 1024 байта. После обработки файла RETRIEVE, выполняет оператор CLOSE в строке 59 и сбрасывает все внутренние буферы. Таким образом, RETRIEVE устанавливает входной файл к началу при каждом запросе извлечения.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* This program reads a name and address data file \*/**

**3 a /\* and displays the information on request. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── retrieve:**

**6 b │ procedure options(main);**

**7 b │**

**8+b │ dcl**

**9+b │ 1 record,**

**10+b │ 2 name character(30) varying,**

**11+b │ 2 addr character(30) varying,**

**12+b │ 2 city character(20) varying,**

**13+b │ 2 state character(10) varying,**

**14+b │ 2 zip fixed decimal(6),**

**15+b │ 2 phone character(12) varying;**

**16 b │**

**17 b │ %replace**

**18 b │ true by '1'b,**

**19 b │ false by '0'b;**

Листинг 8‑6 Программа RETRIEVE

**20 b │**

**21 b │ declare**

**22 b │ (sysprint, input) file,**

**23 b │ filename character(14) varying,**

**24 b │ (lower, upper) character(30) varying,**

**25 b │ eofile bit(1);**

**26 b │**

**27 b │ open file(sysprint) print title('$con');**

**28 b │ put list('Name and Address Retrieval, File Name: ');**

**29 b │ get list(filename);**

**30 b │**

**31 c │ ┌── do while(true);**

**32 c │ │ lower = 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';**

**33 c │ │ upper = 'zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz';**

**34 c │ │ put skip(2) list('Type Lower, Upper Bounds: ');**

**35 c │ │ get list(lower,upper);**

**36 c │ │ if lower = 'EOF' then**

**37 c │ │ stop;**

**38 c │ │**

**39 c │ │ open file(input) stream input environment(b(1024))**

**40 c │ │ title(filename);**

**41 c │ │ eofile = false;**

**42 d │ │ ┌── do while (^eofile);**

**43 d │ │ │ get file(input) list(name);**

**44 d │ │ │ eofile = (name = 'EOF');**

**45 d │ │ │ if ^eofile then**

**46 e │ │ │ ┌── do;**

**47 e │ │ │ │ get file(input)**

**48 e │ │ │ │ list(addr,city,state,zip,phone);**

**49 e │ │ │ │ if name >= lower & name <= upper then**

**50 f │ │ │ │ ┌── do;**

**51 f │ │ │ │ │ put page skip(3)list(name);**

**52 f │ │ │ │ │ put skip list(addr);**

**53 f │ │ │ │ │ put skip list(city,state);**

**54 f │ │ │ │ │ put skip list(zip);**

**55 f │ │ │ │ │ put skip list(phone);**

**56 f │ │ │ │ └── end;**

**57 e │ │ │ └── end;**

**58 d │ │ └── end;**

**59 c │ │ close file(input);**

**60 c │ └── end;**

**61 b │**

**62 b └── end retrieve;**

**Листинг 8-6 Программа RETRIEVE (продолжение)**

*Листинг 8-7* показывает взаимодействие пользователя с программой RETRIEVE. Снова, входной файл - names.dat и существует на диске в форме, созданной CREATE. Входные значения

**B,E**

устанавливает lower значение B и upper значение E и заставляют RETRIEVE, перечислить только Donna Harris. Второй ввод в консоли устанавливает строку lower равной B и upper равной K. Это заставляет RETRIEVE перечислить Donna Harris и John Robinson. Запятая в следующем входном значении устанавливает нижнюю границу в AAA...A и верхнюю границу как K. Таким образом, RETRIEVE перечисляет Arthur Jackson, Donna Harris и John Robinson. Последняя запись состоит только из пары запятых, оставляя нижнюю границу в виде последовательности AAA...A и верхнюю границу в zzz...z. Эти две границы включают весь алфавитный диапазон, поэтому RETRIEVE, отображает весь список имен и адресов. Наконец, ввод EOF заканчивает программу.

В строке 27 *Листинга 8-6* открывается файл SYSPRINT с атрибутом PRINT и title $con (в CP/M). Все файлы рекомендуется открыть с явными атрибутами. В этом случае оператор избыточен, потому что, при выполнении оператора PUT LIST в строке 28, PL/I по умолчанию назначает файлу те же атрибуты.

A>**retrieve**

Name and Address Retrieval, File Name: **names.dat**

Type Lower, Upper Bounds: **B,E**

Donna Harris

2999 Serra Rd.

Chico Ca.

95926

635-3570

Type Lower, Upper Bounds: **B,K**

Donna Harris

2999 Serra Rd.

Chico Ca.

95926

635-3570

John Robinson

805 Franklin St.

Monterey Ca.

93940

649-1000

Type Lower, Upper Bounds: ,K

Arthur Jackson

100 w. 3rd St.

Fresno Ca.

93706

529-1277

Donna Harris

2999 Serra Rd.

Chico Ca.

95926

635-3570

Листинг 8‑7 Взаимодействие с программой RETRIEVE

John Robinson

805 Franklin St.

Monterey Ca.

93940

649-1000

Type Lower, Upper Bounds: **,,**

Arthur Jackson

100 W. 3rd St.

Fresno Ca.

93706

529-1277

Donna Harris

2999 Serra Rd.

Chico Ca.

95926

635-3570

John Robinson

805 Franklin St.

Monterey Ca.

93940

649-1000

Virginia Wilson

?

Unknown Can't Find

99999

?

Type Lower, Upper Bounds; **EOF,,**

A>

**Листинг 8-7 Взаимодействие с программой RETRIEVE (продолжение)**

## Система управления информацией

Следующие четыре примера программ представляют собой модель для системы управления информацией. Эти программы управляют файлом имен сотрудников, адресов, графики оплаты труда и механизмами создания отчетов заработной платы. Каждая из этих программ проста, но вместе они содержат все элементы более развитой системы управления базами данных. Они демонстрируют мощь системы программирования PL/I, обеспечивая основу для пользовательских прикладных программ.

Первая, программа ENTER устанавливает базу данных. Вторая программа, названная KEYFILE, читает эту базу данных и готовит файл ключей для прямого доступа к отдельным записям в базе данных. Третья программа, называемая UPDATE, взаимодействует с пользователем в консоли и позволяет получить доступ к базе данных для поиска и обновления. Наконец, программа REPORT читает базу данных для представления отчета.

### Программа ENTER

*Листинг 8-8* показывает программу ENTER. Программа ENTER взаимодействует с пользователем в консоли и создает начальную базу данных. Основной цикл ввода между строками 40 и 53 запрашивает у пользователя имя сотрудника (name), возраст (age) и почасовую заработную плату (wage). ENTER заполняет структуру данных сотрудника (employee) этой информацией. В строке 48 примера заполняется поле адреса (address) значениями по умолчанию, определенными в структуре в строках с 24 по 33. Вы можете прервать ввод с консоли путем ввода EOF.

Запись employee (сотрудника) содержит несколько полей, общая длина которых составляет 101 байт. Вы можете использовать опцию компилятора $S для проверки этого значения. Оператор OPEN в строке 37 определяет размер фиксированной записи равным 128 байтам, таким образом, позже вы можете расширить записи. Каждая запись файла emp содержит структуру данных только одного сотрудника.

Оператор OPEN назначает emp атрибут KEYED и определяет фиксированный размер каждой записи в опции ENVIRONMENT. Также оператор OPEN определяет размер буфера равным 8000 байтам, которые PL/I автоматически округляет до 8192 байтов. Программа заполняет каждую запись employee (сотрудника) из консольного ввода и записывает запись employee (сотрудника) в файл, получающий название в командной строке, с типом (расширением) файла EMP, строка 38.

Оператор WRITE находится в отдельной подпрограмме, названной WRITE\_IT, начинающейся в строке 55. Размещение кода в отдельной подпрограмме помогает уменьшать размер программы, поскольку программа вызывает WRITE\_IT в двух различных местах, строки 45 и 52.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* This program constructs a data base of employee \*/**

**3 a /\* records using a structure declaration. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a**

**6 a ┌── enter:**

**7 b │ procedure options(main);**

**8 b │ %replace**

**9 b │ true by '1'b,**

**10 b │ false by '0'b;**

**11 b │**

**12 b │ declare**

**13 b │ 1 employee static,**

**14 b │ 2 name character(30) varying,**

**15 b │ 2 address,**

**16 b │ 3 street character(30) varying,**

**17 b │ 3 city character(10) varying,**

**18 b │ 3 state character(12) varying,**

**19 b │ 3 zip fixed decimal(5),**

**20 b │ 2 age fixed decimal(3),**

**21 b │ 2 wage fixed decimal(5,2),**

**22 b │ 2 hours fixed decimal(5,1);**

Листинг 8‑8 Программа ENTER

**23 b │**

**24 b │ declare**

**25 b │ 1 default static,**

**26 b │ 2 street character(30) varying**

**27 b │ initial('(no street)'),**

**28 b │ 2 city character(10) varying**

**29 b │ initial('(no city)'),**

**30 b │ 2 state character(12) varying**

**31 b │ initial('(no state)'),**

**32 b │ 2 zip fixed decimal(5)**

**33 b │ initial(00000);**

**34 b │ declare**

**35 b │ emp file;**

**36 b │**

**37 b │ open file(emp) keyed output environment(f(128),b(8000))**

**38 b │ title ('$1.EMP');**

**39 b │**

**40 c │ ┌── do while(true);**

**41 c │ │ put list('Employee: ');**

**42 c │ │ get list(name);**

**43 c │ │ if name = 'EOF' then**

**44 d │ │ do;**

**45 d │ │ call write\_it();**

**46 d │ │ stop;**

**47 d │ │ end;**

**48 c │ │ address = default;**

**49 c │ │ put list (' Age, Wage: ');**

**50 c │ │ get list (age,wage);**

**51 c │ │ hours = 0;**

**52 c │ │ call write\_it();**

**53 c │ └── end;**

**54 b │**

**55 b │ ┌── write\_it:**

**56 c │ │ procedure;**

**57 c │ │ write file(emp) from(employee);**

**58 c │ └── end write\_it;**

**59 b │**

**60 b └── end enter;**

**Листинг 8-8 Программа ENTER (продолжение)**

*Листинг 8-9* показывает взаимодействие пользователя с программой ENTER, при вводе нескольких записей сотрудника. Ввод EOF завершает программу, закрывает файл plant1.emp и записывает данные на диск.

A>**enter plant1**

Employee: **Jackson**

Age, Wage: **25, 6.75**

Employee: **Harris**

Age, Wage: **30, 9.00**

Листинг 8‑9 Взаимодействие с программой ENTER

Employee: **Robinson**

Age, Wage: **41, 15.00**

Employee: **Wilson**

Age, Wage: **27, 7.50**

Employee: Smith

Age, Wage: **25, ,**

Employee: **Jones**

Age, Wage: **,,**

Employee: **EOF**

A>

Листинг 8-9 Взаимодействие с программой ENTER (продолжение)

### Программа KEYFILE

*Листинг 8-10* показывает программу KEYFILE, которая создает файл ключей, читая файл базы данных, созданный программой ENTER. Файл ключей представляет собой последовательность записей, состоящих из имени сотрудника, за которым следует номер ключа, соответствующего этому имени. В этом случае файл ключей также является потоковым файлом, таким образом, вы можете вывести его на консоль. В строке 16 открывается файл $1.EMP с атрибутом KEYED, определяется длина каждой записи равной 128 байтам и устанавливается размер буфера 10000 байтов. В строке 19 открывается файл ключей, названный keys, как потоковый (STREAM) файл с LINESIZE(60) и опцией TITLE, которая добавляет KEY в качестве типа (расширения) файла.

В строке 23, программа KEYFILE читает последовательные записи, извлекает ключ с опцией KEYTO и записывает имя и ключ на консоль и в файла ключей. Пример взаимодействия в *Листинге 8-11* иллюстрирует вывод из KEYFILE, используя базу данных plant1.emp. Каждое значение ключа, извлеченное оператором READ, является относительным номером записи, соответствующим позиции записи в файле.

После выполнения программы KEYFILE вы можете использовать команду

A>**type plant1.key**

для отображения содержания файла plant1.key, как показано в *Листинге 8-12*.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* This program reads an employee record file and \*/**

**3 a /\* creates another file of keys to access the records.\*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a**

**6 a ┌── keyfile:**

**7 b │ procedure options(main);**

**8 b │ declare**

**9 b │ 1 employee static,**

**10 b │ 2 name character(30) varying;**

**11 b │**

**12 b │ declare**

**13 b │ (input, keys) file,**

**14 b │ k fixed;**

**15 b │**

**16 b │ open file(input) keyed environment(f(128),b(10000))**

**17 b │ title('$1.emp');**

**18 b**

**19 b │ open file(keys) stream output**

**20 b │ linesize (60) title('$1.key');**

**21 b │**

**22 c │ ┌── do while('1');**

**23 c │ │ read file(input) into(employee) keyto(k);**

**24 c │ │ put skip list(k,name);**

**25 c │ │ put file(keys) list(name,k);**

**26 c │ │ if name = 'EOF' then**

**27 c │ │ stop;**

**28 c │ └── end;**

**29 b │**

**30 b └── end keyfile;**

Листинг 8‑10 Программа KEYFILE

A>**keyfile plant1**

0 Jackson

1 Harris

2 Robinson

3 Wilson

4 Smith

5 Jones

6 EOF

A>

Листинг 8‑11 Взаимодействие с программой KEYFILE

A>**type plant1.key**

'Jackson' 0 'Harris' 1 'Robinson' 2

'Wilson' 3 'Smith' 4 'Jones' 5 'EOF'

6

A>

Листинг 8‑12 Содержимое файла ключей

### Программа UPDATE

Программа UPDATE в *Листинге 8-13* позволяет, получить доступ к базе данных, созданной программой ENTER и индексируемой с помощью файла, созданного KEYFILE. Программа UPDATE сначала читает потоковый файл .key, в структуру данных, названную keylist. Keylist связывает имя сотрудника с соответствующим значением ключа в базе данных. Строки 20 - 23 объявляют структуру данных, которая содержит значения этих ссылок, и в строках 37 - 40 происходит заполнение данных.

**Примечание**: Оператор в строке 39 не является многократным присваиванием, а скорее определением Булевого выражения для переменной eolist.

В строке 31 UPDATE открывает файл .emp. Оператор OPEN присваивает файлу атрибут DIRECT, который позволяет операции READ и WRITE с индивидуальными записями, идентифицируемыми значением ключа. Затем вы вводите имя сотрудника как matchname, и DO-группа между строками 47 и 61 напрямую обращается к отдельным записям в базе данных.

Прямой доступ происходит следующим образом. В строке 48 осуществляется поиск списка имен, считанных из файла .key. Если есть соответствие, оператор READ с KEY в строке 50 загружает запись employee (сотрудника) в память из файла .emp. Программа отображает и обновляет различные поля с консоли, и затем переписывает запись в базу данных с помощью оператора WRITE с KEYFROM в строке 58. UPDATE завершает выполнение, когда вы вводите EOF.

*Листинг 8-14* показывает три последовательных сеанса обновления, во время которых обновляются различные адреса и рабочее время. В каждом сеансе вы вводите имя сотрудника, осуществляете доступ и отображаете запись, и необязательно, обновляете поля. Здесь полезен оператор GET LIST. Чтобы изменить значение, вы просто вводите новое значение в позиции поля. Если вы не хотите изменять значение, ввод разделителя запятой оставляет поле неизменным.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* This program allows you to retrieve and update \*/**

**3 a /\* individual records in an employee data base using \*/**

**4 a /\* a keyed file. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── update:**

**7 b │ procedure options(main);**

**8 b │ declare**

**9 b │ 1 employee static,**

**10 b │ 2 name character(30) varying,**

**11 b │ 2 address,**

**12 b │ 3 street character(30) varying,**

**13 b │ 3 city character(10) varying,**

**14 b │ 3 state character(12) varying,**

**15 b │ 3 zip fixed decimal(5),**

**16 b │ 2 age fixed decimal(3),**

**17 b │ 2 wage fixed decimal(5,2),**

**18 b │ 2 hours fixed decimal(5,1);**

**19 b │**

**20 b │ declare**

**21 b │ 1 keylist (100),**

**22 b │ 2 keyname character(30) varying,**

**23 b │ 2 keyval fixed binary;**

**24 b │**

**25 b │ declare**

**26 b │ (i, endlist) fixed,**

**27 b │ eolist bit(1) static initial('0'b),**

**28 b │ matchname character(30) varying,**

**29 b │ (emp, keys) file;**

**30 b │**

**31 b │ open file(emp) update direct environment(f(128))**

**32 b │ title ('$1.EMP');**

**33 b │**

**34 b │ open file(keys) stream environment(b(4000))**

**35 b │ title('$1.key');**

**36 b │**

Листинг 8‑13 Программа UPDATE

**37 c │ ┌── do i = 1 to 100 while(^eolist);**

**38 c │ │ get file(keys) list(keyname(i),keyval(i));**

**39 c │ │ eolist = keyname(i) = 'EOF';**

**40 c │ └── end;**

**41 b │**

**42 c │ ┌── do while('1'b);**

**43 c │ │ put skip list('Employee: ');**

**44 c │ │ get list(matchname);**

**45 c │ │ if matchname = 'EOF' then**

**46 c │ │ stop;**

**47 d │ │ ┌── do i = 1 to 100;**

**48 d │ │ │ if matchname = keyname(i) then**

**49 e │ │ │ ┌── do;**

**50 e │ │ │ │ read file(emp) into(employee)**

**51 e │ │ │ │ key(keyval(i));**

**52 e │ │ │ │ put skip list('Address: ',**

**53 e │ │ │ │ street, city, state, zip);**

**54 e │ │ │ │ put skip list(' ');**

**55 e │ │ │ │ get list(street, city, state, zip);**

**56 e │ │ │ │ put list('Hours:',hours,': ');**

**57 e │ │ │ │ get list(hours);**

**58 e │ │ │ │ write file(emp) from (employee)**

**59 e │ │ │ │ keyfrom(keyval(i));**

**60 e │ │ │ └── end;**

**61 d │ │ └── end;**

**62 c │ └── end;**

**63 b │**

**64 b └── end update;**

**Листинг 8-13 Программа UPDATE (продолжение)**

A>**update plant1**

Employee: **Jackson**

Address: (no street) (no city) (no state) 0

**'100 W. 3rd St.', 'Fresno', 'Ca.', 93706**

Hours: 0.0 : **40.0**

Employee: **EOF**

A>**update plant1**

Employee: **Harris**

Address: (no street) (no city) (no state) 0

**'2999 Serra Rd.', 'Chico', 'Ca. ', 95926**

Hours: 0.0 : **46.0**

Employee: **EOF**

Листинг 8‑14 Взаимодействие с программой UPDATE

A>**update plant1**

Employee: **Harris**

Address: 2999 Serra Rd. Chico Ca. 95926

**,,,,**

Hours: 46.0 : **48.0**

Employee: **EOF**

A>**update plant1**

Employee: **Wilson**

Address: (no street) (no city) (no state) 0

**,,,,**

Hours: 0.0 : **35.5**

Employee: **EOF**

A>**update plant1**

Employee: **Wilson**

Address: (no street) (no city) (no state) 0

**'556 Palm Ave.', 'Burbank', 'Ca.', 91507**

Hours: 35.5 : **,**

Employee: **EOF**

A>

**Листинг 8-14 Взаимодействие с программой UPDATE (продолжение)**

### Программа REPORT

*Листинг 8-15* показывает программу REPORT. Программа REPORT использует обновленный файл сотрудников, чтобы произвести список сотрудников вместе с значениями их зарплат. Программа REPORT также получает доступ к файлу сотрудников, но она читает файл последовательно, для создания желаемого вывода. Основная DO-группа между строками 35 и 51 считывает каждую последовательную запись сотрудника и конструирует строку заголовка форму

**[name]**

сопровождаемую суммой в долларах. REPORT использует потоковую форму оператора WRITE, строки 41 и 50, чтобы продолжить выходную линию. Строка 40 включает встроенные управляющие символы ^M и ^J в конце buff, чтобы вызвать возврат каретки и перевод строки при записи буфера. Затем программа REPORT вычисляет значение зарплаты и присваивает его строке CHARACTER VARYING, названной buff в строке 44. В этом присвоении PL/I выполняет автоматическое преобразование данных из FIXED DECIMAL в CHARACTER с начальными пробелами. REPORT также сканирует начальные пробелы, заменяя их последовательностью тире знак доллара, чтобы выровнять вывод, и записывает данные в файл отчета.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* This program reads an employee data base and \*/**

**3 a /\* prints a list of paychecks. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── report:**

**6 b │ procedure options(main);**

**7 b │ declare**

**8 b │ 1 employee static,**

**9 b │ 2 name character(30) varying,**

**10 b │ 2 address,**

**11 b │ 3 street character(30) varying,**

**12 b │ 3 city character(10) varying,**

**13 b │ 3 state character(12) varying,**

**14 b │ 3 zip fixed decimal(5),**

**15 b │ 2 age fixed decimal(3),**

**16 b │ 2 wage fixed decimal(5,2),**

**17 b │ 2 hours fixed decimal(5,1);**

**18 b │**

**19 b │ declare**

**20 b │ i fixed,**

**21 b │ dashes character(15) static initial**

**22 b │ ('$--------------'),**

**23 b │ buff character(20) varying,**

**24 b │ (grosspay, withhold) fixed decimal(7,2),**

**25 b │ (repfile, empfile) file;**

**26 b │**

**27 b │ open file(empfile) keyed environment(f(128),b(4000))**

**28 b │ title ('$1.EMP');**

**29 b │ open file(repfile) stream print environment(b(2000))**

**30 b │ title('$2.$2');**

**31 b │**

**32 b │ put list('Set Top of Forms, Press Return');**

**33 b │ get skip;**

**34 b │**

**35 c │ ┌── do while('1'b);**

**36 c │ │ read file(empfile) into(employee);**

**37 c │ │ if name = 'EOF' then**

**38 c │ │ stop;**

**39 c │ │ put file(repfile) skip(2);**

**40 c │ │ buff = '[' !! name !! ']^m^j';**

**41 c │ │ write file(repfile) from (buff);**

**42 c │ │ grosspay = wage \* hours;**

**43 c │ │ withhold = grosspay \* .15;**

**44 c │ │ buff = grosspay - withhold;**

**45 d │ │ ┌── do i = 1 to 15**

**46 d │ │ │ while (substr(buff,i,1) = ' ');**

**47 d │ │ └── end;**

**48 c │ │ i = i - 1;**

**49 c │ │ substr(buff,1,i) = substr(dashes,1,i);**

**50 c │ │ write file (repfile) from(buff);**

**51 c │ └── end;**

**52 b │**

**53 b └── end report;**

Листинг 8‑15 Программа REPORT

*Листинги 8-16* и *8-17* показывают вывод из программы REPORT. В первом случае, команда

A>**report plantl $con**

отправляет отчет на консоль для анализа. Во втором случае, команда

A>**report plant1 plant1.prn**

отправляет вывод в дисковый файл plant1.prn. Затем вы можете исследовать содержание файла с помощью команды:

A>**type plant1.prn**

A>**report plant1 $con**

Set Top of Forms, Press Return

[Jackson]

$----229.50

[Harris]

$----367.20

[Robinson]

$------0.00

[Wilson]

$----226.32

[Smith]

$------0.00

[Jones]

$------0.00

A>

Листинг 8‑16 Вывод REPORT на консоль

A>**report plant1 plant1.prn**

Set Top of Forms, Press Return

A>**type plant1.prn**

[Jackson]

$----229.50

[Harris]

$----367.20

[Robinson]

$------0.00

[Wilson]

$----226.32

[Smith]

$------0.00

[Jones]

$------0.00

Листинг 8‑17 Вывод REPORT в файл на диске

Ссылки: Справочное руководство Разделы 10.1, 10.8, 11.2, 12

**Конец раздела 8**

# Константы, переменные и параметры типа метки

Каждая из программ, представленных до сих пор, заканчивает выполнение обнаружив состояние конца файла с соответствующим трассировкой ENDFILE, или при помощи специального значения данных, которое сигнализирует условие конца данных. Программы DECPOLY и FLTPOLY обнаруживают условие конца данных, проверяя особый случай, когда все три входных значения х, у и z равны нулю.

К счастью, PL/I обеспечивает более изящные способы обнаружения состояния конца данных. На самом деле обнаружение условия конца данных является только одним из многих направлений обработки состояний. Чаще всего обработка этих состояний включает помеченные операторы. Вам понадобится некоторый опыт в обработке меток, прежде чем браться за общую тему обработки состояний в [*Разделе 10*](#_Обработка_состояний).

## Помеченные операторы

Избежание использования помеченных операторов и GOTO, приводящих в результате к неструктурированным программам является аксиомой программирования. Программы, содержащие много помеченных операторов, часто трудны для понимания другими программистами. Такие программы становятся нечитаемыми, даже автором, по мере увеличения размера программы.

PL/I поощряет хорошую структуру, предоставляя полный набор управляющих структур в форме итеративных DO-групп с опциями REPEAT и WHILE. Эти управляющие структуры исключает необходимость помеченных операторов в общей схеме программирования. Вы должны использовать эти управляющие структуры, когда это возможно, и ограничивать использование помеченных операторов при обработке состояний и локально определенных вычисляемых GOTO.

Разумное использование помеченных операторов целесообразно при обработке состояния. Возникновение ошибки, такой как ввод входной строки данных с опечаткой, легко обрабатывается, передачей программного управления метке во внешнем блоке, в котором происходит восстановление. Этот метод понимания хода выполнения программы проще, чем обычная система флагов, тестов и операторов возврата.

## Метки программы

Метки программы, как и другие типы данных PL/I, делятся на две широкие категории: константы и переменные типа метки. Константа типа метки появляется литерально в исходной программе, и ее значение не изменяется во время работы программы. Переменная типа метки не имеет начального значения, и вы должны присвоить ей значение константы типа метки с помощью прямого оператор присваивания, или с помощью неявного присвоения параметра в вызове подпрограммы.

Следующая последовательность кода является примером константы типа метки, предшествующей оператору PL/I.

**on error(1)  
 begin;  
 put skip list('Bad Input, Try Again');  
 goto retry;  
 end;  
...  
retry: get list(name);**

**...**

Оператор on error(1) устанавливает ловушку для определенного состояния. Если состояние возникает из-за недопустимого ввода, то управление передается в блок BEGIN, который выводит сообщение об ошибке, и затем возвращает управление к помеченному оператору. Если при вводе ошибки нет, управление передается следующему оператору после оператора GET LIST.

## Вычисляемый GOTO

В PL/I константа типа метки может содержать единственный положительный или отрицательный литеральный индекс. Преобразованная в индекс константа типа метки соответствует цели n-пути ответвления, т.е. вычисляемый GOTO. Следующая последовательность кода показывает конкретный пример.

**get list(x);  
 goto q(x);  
q(-1):  
 y = f1(x);  
 goto endq;  
q(0):  
 y = f2(x);  
 goto endq;  
q(2):;  
q(3):  
 y = f3(x);  
endq:  
put skip list('f(x)=',y);  
...**

Этот код неявно определяет четыре константы типа метки: q(-1), q(0), q(2) и q(3). Компилятор автоматически определяет внутренний вектор констант типа меток

**q(-1:3) label constant**

содержащий значения этих констант типа меток.

Предыдущий оператор не является допустимым оператором PL/I, но указывает, что компилятор делает внутренне когда обнаруживает такие операторы в исходном коде. Кроме того, при использовании таких конструкций управление не передается индексу, не имеющего значения соответствующей константы типа метки. В предыдущем примере ответвление к q(1) приводит к неопределенным результатам.

## Ссылки на метку

Ссылка на константу типа метки может быть локальной или нелокальной. *Локальная* ссылка на константу типа метки означает, что метка встречается как цель оператора GOTO только в блоке PROCEDURE или BEGIN, который содержит GOTO. *Нелокальная* ссылка на константу типа метки означает, что метка встречается в правой части присваивания переменной типа метки, в качестве аргумента функции, или как цель оператора GOTO во внутреннем вложенном блоке PROCEDURE или BEGIN.

Хотя функционально нет никакой разницы между обработкой локальной и нелокальной ссылки на константу типа метки, нелокальная ссылка требует дополнительное пространство и время. По этой причине PL/I предполагает, что на индексируемую константу типа метки ссылка будет только локальной. Если программа передает управление на индексируемую константу типа метки из-за пределов текущей среды, возможны неопределенные результаты.

В качестве примера, рассмотрим следующую последовательность кода:

**┌── main:  
│**  **procedure options(main);  
│** **┌── P1:  
│** **│ procedure;  
│** **│ goto lab1;  
│** **│ goto lab2;  
│** **│ ┌── P2:  
│** **│ │ procedure;  
│** **│ │ goto lab2;  
│** **│ └── end P2;  
│** **│ lab1:;  
│** **│ lab2:;  
│** **└── end P1;  
└── end main;**

На константу типа метки lab1 имеется только локальная ссылка в процедуре P1, в то время как lab2 является целью локальной ссылки в P1 и нелокальной ссылки в P2.

## Пример программы

*Листинг 9-1* показывает нефункциональную программу, которая иллюстрирует использование различных констант и переменных типа меток. Константами типа метки в программе LABELS являются c(1), c(2), c(3), lab1 и lab2. Они определены их литеральным появлением в программе. Переменными типа метки являются x, y, z и g, и определены объявлениями в строках 10 и 38.

В начале выполнения переменные типа метки имеют неопределенные значения. Программа сначала присваивает переменной x значение константы lab1. Затем переменная типа метки y, косвенно получает значение константы lab1 посредством присвоения в строке 12. В результате все три оператора GOTO в строках 14, 15 и 16 функционально эквивалентны. Каждый оператор передает управление пустому оператору после метки lab1 в строке 32.

Вызов подпрограммы в строке 18 показывает другую форму присвоения переменной. Lab2 является аргументом, отправленным в процедуру P и присваивается формальной переменной метки g. В этой программе вызов подпрограммы передает программное управление непосредственно оператору помеченному lab1.

DO-группа, начинающая в строке 20, инициализирует вектор переменных типа меток z соответствующими значениями вектора констант типа меток c. Благодаря этой инициализации, два вычисляемый оператор GOTO, начиная со строки 25, имеют одинаковый эффект.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* This is a non-functional program. Its purpose is \*/**

**3 a /\* to illustrate the concept of label constants and \*/**

**4 a /\* variables. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── Labels:**

**7 b │ procedure options(main);**

**8 b │ declare**

**9 b │ i fixed,**

**10 b │ (x, y, z(3)) label;**

**11 b │ x = lab1;**

**12 b │ y = x;**

**13 b │**

**14 b │ goto lab1;**

**15 b │ goto x;**

**16 b │ goto y;**

**17 b │**

**18 b │ call P(lab2);**

**19 b │**

**20 c │ ┌── do i = 1 to 3;**

**21 c │ │ z(i) = c(i);**

**22 c │ └── end;**

**23 b │**

**24 b │ i = 2;**

**25 b │ goto z(i);**

**26 b │ goto c(i);**

**27 b │**

**28 b │ c(1):;**

**29 b │ c(2):;**

**30 b │ c(3):;**

**31 b │**

**32 b │ lab1:;**

**33 b │ lab2:;**

**34 b │**

**35 b │ ┌── P:**

**36 c │ │ procedure (g);**

**37 c │ │ declare**

**38 c │ │ g label;**

**39 c │ │ goto g;**

**40 c │ └── end P;**

**41 b │**

**42 b └── end Labels;**

Листинг 9‑1 Иллюстрация переменных и констант типа меток

Ссылки: Справочное руководство разделы 3.3, 8.5, 8.6

**Конец раздела 9**

# Обработка состояний

Обработка состояний является важной функциональной возможностью любого промышленного языка программирования. Язык должен позволить программе прерывать и обрабатывать состояние ошибки периода выполнения с помощью определенных действий программы, и затем продолжать выполнение.

Например, общее состояние возникает, когда программа считывает входные данные из интерактивной консоли, и вы случайно вводите значение, которое не соответствует типу данных входной переменной. Система времени выполнения PL/I сигнализирует об ошибке преобразования, и при отсутствии каких-либо определенных программой действий, заканчивает программу трассировкой. Если это преждевременное завершение происходит после нескольких часов ввода данных, оно приводит к непроизводительной трате усилий. Это недопустимо в производственной среде.

## Категории состояний

PL/I обеспечивает девять категорий состояний. Ими являются:

|  |  |
| --- | --- |
| * ERROR * FIXEDOVERFLOW * OVERFLOW * UNDERFLOW * ZERODIVIDE | * ENDFILE * UNDEFINEDFILE * KEY * ENDPAGE |

Первые пять категорий включают все состояния арифметических ошибок и разные состояния, возникающие во время установки и обработки ввода-вывода. Они также включают ошибки преобразования между различными типами данных. Последние четыре категории применяются к определенному файлу, к которому получает доступ система ввода-вывода. Каждое состояние имеет соответствующий подкод, который предоставляет информацию об источнике состоянии.

## Операторы обработки состояний

В PL/I обработку состояний осуществляют операторы ON, REVERT и SIGNAL. Оператор ON определяет действия, которые происходят после обнаружения состояния. Оператор REVERT отключает оператор ON. Оператор SIGNAL позволяет программе сигнализировать различные состояния.

### Операторы ON и REVERT

Следующий фрагмент программы иллюстрирует операторы ON и REVERT в DO-группе.

**┌── do while(^EOF);  
│ on endfile(sysin)  
│ EOF = '1'b;**

**│ ...  
│ revert endfile(sysin);  
└── end;**

Здесь, операторы ON и REVERT выполняются на каждой итерации. Обработка операторов ON и REVERT включает накладные расходы на выполнение.

Чтобы избежать этого, закодируйте ту же DO-группу следующим образом:

**on endfile(sysin)  
 EOF = 'true';  
 EOF = 'false';  
┌── do while(^EOF);**

**│ ...  
└── end;**

PL/I автоматически выполняет оператор REVERT для любого ON состояния, которое вы включаете в блоке процедуры, когда управление переходит за пределы блока. Программа, показанная в *Листинге 10-1* иллюстрирует это понятие.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* This program is nonfunctional. Its purpose is to \*/**

**3 a /\* illustrate how PL/I executes the ON and REVERT \*/**

**4 a /\* statements. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── auto\_revert:**

**7 b │ procedure options(main);**

**8 b │ declare**

**9 b │ i fixed,**

**10 b │ sysin file;**

**11 b │**

**12 c │ ┌── do i = 1 to 10000;**

**13 c │ │ call p(i,exit);**

**14 c │ │ exit:**

**15 c │ └── end;**

**16 b │**

**17 b │ ┌── P:**

**18 c │ │ procedure (index,lab);**

**19 c │ │ declare**

**20 c │ │ (t, index) fixed,**

**21 c │ │ lab label;**

**22 c │ │**

**23 c │ │ on endfile(sysin)**

**24 c │ │ goto lab;**

**25 c │ │**

**26 c │ │ put skip list(index,':');**

**27 c │ │ get list(t);**

**28 c │ │ if t = index then**

**29 c │ │ goto lab;**

**30 c │ └── end P; /\* Здесь подразумевается неявный REVERT \*/**

**31 b │**

**32 b └── end auto\_revert;**

Листинг 10‑1 Программа REVERT

В программе REVERT в строке 13 вызывается процедура P и в нее передаются фактические параметры: индекс DO-группы i и константа типа метки exit. Оператор ON в P выполняется при каждом вызове процедуры. Таким образом REVERT, имеет три возможных пути выхода из процедуры P.

Если вы вводите символ конца файла Ctrl-Z, REVERT выполняет включенное ON состояние и передает управление с помощью переменной типа метка lab оператору помеченному exit. PL/I деактивирует процедуру и выполняет оператор REVERT, поскольку оператор GOTO передает управление вне среды P.

Второй возможный выход следует по результату проверки в строке 28. Если вы вводите значение, равное индексу, то выполняется оператор GOTO в строке 29 и снова управление передается вне среды P.

И наконец, если управление достигает конца P, PL/I выполняет оператор REVERT и отключает ON состояние в строке 23. Независимо от того как управление покидает среду процедуры, PL/I всегда отключает состояние ON.

### Оператор SIGNAL

Оператор SIGNAL активирует ON-тело, операторы тела, соответствующие конкретному оператору ON. Таким образом обработка оператора SIGNAL имеет тот же эффект, как и при активизации состояния системой времени выполнения.

Следующий фрагмент программы иллюстрирует оператор SIGNAL.

**on endfile(sysin)  
 stop;**

**do while ('l'b)  
 get list(buff);  
 if buff = 'END' then  
 signal endfile(sysin);  
 put skip list(buff);**

**end;**

**...**

Этот код выполняет оператор SIGNAL каждый раз, когда оператор GET LIST читает значение 'END' из файла SYSIN. Таким образом, ON состояние получает управление при фактическом достижении конца файла или когда прочитано значение 'END'.

## Примеры обработки состояний

Следующие две программы: FLTPOLY2 и COPYLPT, включают обработку некоторых состояний, таким образом, вы можете увидеть, как реализуются эти понятия.

### Программа FLTPOLY2

Листинг 10-2 показывает программу FLTPOLY2. Это - по существу та же программа, приведенная в *Разделе 7-1*. Единственное отличие состоит в том, что она включает в себя обработку состояния для перехвата состояния конца-файла для файла SYSIN. Если вы запустите эту программу, то увидите, что вы можете остановить ее с помощью символа Ctrl-Z. В отличие от FLTPOLY, если вы вводите все нули, FLTPOLY2 просто вычисляет многочлен и запрашивает ввод.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет полиномиальное выражение, \*/**

**3 a /\* используя данные FLOAT BINARY. Она также перехва- \*/**

**4 a /\* тывает состояние конец файла для файла SYSIN. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── fltpoly2:**

**7 b │ procedure options(main);**

**8 b │ %replace**

**9 b │ false by '0'b,**

**10 b │ true by '1'b;**

**11 b │ declare**

**12 b │ (x,y,z) float binary(24),**

**13 b │ eofile bit(1) static initial(false),**

**14 b │ sysin file;**

**15 b │**

**16 b │ on endfile(sysin)**

**17 b │ eofile = true;**

**18 b │**

**19 c │ ┌── do while(true);**

**20 c │ │ put skip(2) list('Type x,y,z: ');**

**21 c │ │ get list(x,y,z);**

**22 c │ │**

**23 c │ │ if eofile then**

**24 c │ │ stop;**

**25 c │ │**

**26 c │ │ put skip list(' 2');**

**27 c │ │ put skip list(' x + 2y + z =',P(x,y,z));**

**28 c │ └── end;**

**29 b │**

**30 b │ ┌── P:**

**31 c │ │ procedure (x,y,z) returns (float binary(24));**

**32 c │ │ declare**

**33 c │ │ (x,y,z) float binary(24);**

**34 c │ │ return (x \* x + 2 \* y + z);**

**35 c │ └── end P;**

**36 b │**

**37 b └── end fltpoly2;**

Листинг 10‑2 Программа FLTPOLY2

### Программа COPYLPT

*Листинг 10-3* показывает пример использования обработки ввода-вывода с помощью ON состояний. Программа COPYLPT копирует потоковый файл (STREAM) с диска в файла печати (PRINT), правильно форматируя выходную строку с верхним колонтитулом страницы и номерами строк. Программа принимает входные данные консоли для получения параметров операции копирования и обеспечивает выходы при ошибках и повторные операций для каждого входного значения. COPYLPT устанавливает различные ON-блоки для перехвата ошибки во время операции копирования, которая происходит в итеративной DO-группе между строками 71 и 76. Следующие разделы обсуждают отдельные части программы.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа копирует потоковый файл на диске в \*/**

**3 a /\* файл печати и форматирует вывод с верхним колон- \*/**

**4 a /\* титулом страницы и номерами строк. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 b ┌── copy: procedure options(main);**

**7 b │**

**8 b │ declare**

**9 b │ (sysin, sourcefile, printfile) file,**

**10 b │ (pagesize, pagewidth, spaces, linenumber) fixed,**

**11 b │ (line character(14), buff character(254)) varying;**

**12 b │**

**13 b │ put list('^z File to Print Copy Program');**

**14 b │**

**15 b │ on endfile(sysin)**

**16 b │ go to typeover;**

**17 b │**

**18 b │ typeover:**

**19 b │ put skip(5) list('How Many Lines Per Page? ');**

**20 b │ get list(pagesize);**

**21 b │**

**22 b │ put skip list('How Many Column Positions? ');**

**23 b │ get skip list(pagewidth);**

**24 b │**

**25 b │ on error(1)**

**26 c │ ┌── begin;**

**27 c │ │ put list('Invalid Number, Type Integer');**

**28 c │ │ go to getnumber;**

**29 c │ └── end;**

**30 b │ getnumber:**

**31 b │ put skip list('Line Spacing (1=Single)? ');**

**32 b │ get skip list(spaces);**

**33 b │ revert error(1);**

**34 b │**

**35 b │ put skip list('Destination Device/File: ');**

**36 b │ get skip list(line);**

**37 b │**

**38 b │ open file(printfile) print pagesize(pagesize)**

**39 b │ linesize(pagewidth) title(line);**

**40 b │**

**41 b │ on undefinedfile(sourcefile)**

**42 c │ ┌── begin;**

**43 c │ │ put skip list('"',line,'" isn''t a Valid Name');**

**44 c │ │ go to retry;**

**45 c │ └── end;**

**46 b │ retry:**

**47 b │ put skip list('Source File to Print? ');**

**48 b │ get list(line);**

**49 b │ open file(sourcefile) stream environment(b(8000))**

**50 b │ title(line);**

Листинг 10‑3 Программа COPYLPT

**51 b │ on endfile(sourcefile)**

**52 c │ ┌── begin;**

**53 c │ │ put file(printfile) page;**

**54 c │ │ stop;**

**55 c │ └── end;**

**56 b │**

**57 b │ on endfile(printfile)**

**58 c │ ┌── begin;**

**59 c │ │ put skip list('^g^g^g^g Disk is Full');**

**60 c │ │ stop;**

**61 c │ └── end;**

**62 b │**

**63 b │ on endpage(printfile)**

**64 c │ ┌── begin;**

**65 c │ │ put file(printfile) page skip(2)**

**66 c │ │ list('PAGE',pageno(printfile));**

**67 c │ │ put file(printfile) skip(4);**

**68 c │ └── end;**

**69 b │**

**70 b │ signal endpage(printfile);**

**71 c │ ┌── do linenumber = 1 repeat(linenumber + 1);**

**72 c │ │ get file (sourcefile) edit(buff) (a);**

**73 c │ │ put file (printfile)**

**74 c │ │ edit(linenumber,'|',buff) (f(5),x(1),a(2),a);**

**75 c │ │ put file (printfile) skip(spaces);**

**76 c │ └── end;**

**77 b │**

**78 b └── end copy;**

Листинг 10-3 Программа COPYLPT (продолжение)

Программа COPYLPT начинается, читая пять значений:

* количество строк на каждой странице;
* ширина строки принтера;
* межстрочный интервал, обычно одинарный или двойной интервал вывода;
* целевой файл или устройство;
* файл источника или устройство.

При вводе этих параметров вы можете ввести символ конца файла Ctrl-Z и перезапустить запрос.

Оператор PUT LIST в строке 13 записывает первоначальное сообщение входа в систему. Напомним, что PL/I позволяет управляющие символы в строковых константах. Здесь, первый символ сообщения - Ctrl-Z, который очищает экран, если вы используете в качестве CRT устройство ADM-3A™. Если вы используете некоторое другое устройство, вы можете заменить его надлежащим символом и перекомпилировать программу.

Оператор ON строки 15 прерывает состояние ENDFILE для файла SYSIN, так, чтобы выполнение печати нового текста началось поверх старого каждый раз, когда консоль читает символ конца файла.

В строках 19 - 23 осуществляется чтение первых двух параметров без проверки ошибок, кроме обнаружения конца файла. Строка 25, однако, перехватывает ошибки преобразования для всех последующих операций. Если оператор GET в строке 32 считывает нечисловое поле, управление передается в ON-тело, расположенное между строками 26 и 29, которое пишет сообщение об ошибке, переходит к getnumber, и повторяет операцию ввода. После успешного ввода параметров пространства оператор REVERT в строке 33 отключает обработку ошибок преобразования.

COPYLPT открывает входной и выходной файлы между строками 38 и 50. Программа предполагает, что выходной файл всегда можно открыть, но обнаруживает неопределенный (UNDEFINED) входной файл, таким образом, вы можете исправить имя файла.

Программа выполняет два оператора ON ENDFILE между строками 51 и 61. Первый оператор перехватывает входное состояние конца файла и выполняет извлечение страницы, в выходном файле. Это гарантирует, вывод на принтер сверху новой страницы после завершения работы печати. Оператор STOP, включенный в ON-блок, завершает обработку с выходом.

Второй ON-блок перехватывает состояние конца файла в файле печати. Это может произойти только, если дисковый файл заполняется, поэтому, блок печатает сообщение

**Disk is Full**

и выполнение завершается. Символ Ctrl-G отправляет несколько звуковых сигналов на CRT в качестве аварийного сигнала. Система времени выполнения закрывает все файлы после завершения с тем, чтобы файл печати был без повреждений до полной емкости диска.

В строка 63 начинается блок ON ENDPAGE, который прерывает состояние конца страницы для файла печати. Каждый раз, когда система времени выполнения устанавливает это состояние, ON блок перемещает к верху следующей страницы, пропускает две строки, распечатывает номер страницы и пропускает еще четыре строки прежде, чем вернуться к источнику вызвавшему состояние. Оператор SIGNAL в строке 70 начинает вывод файла печати с новой страницы, передавая управление в ON блок, определенный в строке 63. Все последующие состояния ENDPAGE генерируются системой времени выполнения в конце каждой страницы.

DO-группа, начинающаяся в строке 71, инициализирует и увеличивает на 1 счетчик строки в каждой итерации. Оператор GET EDIT в строке 72 определяет алфавитно-цифровой формат A. Она заполняет буфер следующей входной строкой до, но не включая, последовательность символов возврата каретки/перевода строки. Оператор PUT EDIT в строке 73 записывает строку в файл назначения с предшествующими номером строки, пробелом, вертикальной полосой и следующим пробелом, взятом из поля A(2). Если система времени выполнения устанавливает состояние ENDPAGE при выполнении оператора PUT в строке 75, элемент формата SKIP(spaces) не может быть обработан.

*Листинг 10-*4 показывает взаимодействие пользователя с программой COPYLPT. Здесь, исходный файл - программа COPY.PLI, и физический принтер $LST является местом назначения.

A>**copylpt**

File to Print Copy Program

How Many Lines Per Page? **26**

How Many Column Positions? **80**

Line Spacing (1=Single)? **Yes**

Invalid Number, Type Integer

Line Spacing (1=Single)? **1**

Destination Device/File: **$lst**

Source File to Print? copy.pil

" copy.pil " isn't a Valid Name

Source File to Print? copy.pli

Листинг 10‑4 Взаимодействие с COPYLPT

*Листинг 10-5* показывает две страницы вывода, произведенные программой.

PAGE 1

1 | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

2 | /\* This program copies one file to another using \*/

3 | /\* buffered I/O. \*/

4 | /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

5 | copy:

6 | procedure options(main);

7 | declare

8 | (input\_file,output\_file) file;

9 |

10 | open file (input\_file) stream

11 | environment(b(8192)) title('$1.$1');

12 |

13 | open file (output\_file) stream output

14 | environment(b(8192)) title('$2.$2');

15 | declare

16 | buff character(254) varying;

17 |

18 | do while('1'b);

19 | read file (input\_file) into (buff);

20 | write file (output\_file) from (buff);

PAGE 2

21 | end;

22 | end copy;

Листинг 10‑5 Вывод программы COPYLPT

Этот пример показывает, как вы можете включить обработку ошибок в свои программы, чтобы сделать их проще в использовании. На самом деле вы можете улучшить программу COPYLPT добавив обработку ошибок в первых двух строках ввода, или ошибок в имени файла назначения.

Для получения дополнительного опыта, вы можете вернуться ко всем предыдущим примерам и добавить ON-блоки для перехвата недопустимых входных данных и состояний конца файла. Изменение этих небольших программ даст вам хорошую основу в обработке состояний.

Ссылки: Справочное руководство разделы с 9.1 по 9.3, 10.5, 11.3

**Конец раздела 10**

# Обработка символьной строки

PL/I обеспечивает мощные возможности обработки символьной строки, важные в коммерческом промышленном языке. Этот раздел представляет два примера программ, которые иллюстрируют использование некоторых строковых функций PL/I. После того, как вы прочитаете текст и изучите примеры программ, вы можете внести изменения в программы, чтобы расширить свои знания PL/I.

## Программа OPTIMIST

Наш первый пример обработки строк - программа под названием OPTIMIST. Программа OPTIMIST превращает отрицательное предложение в положительное предложение. OPTIMIST выполняет эту задачу при помощи средств символьной строки PL/I.

*Листинг 11-1* показывает программу OPTIMIST. Первый сегмент, между строками 12 и 23, определяет элементы данных, используемые в программе. Остальная часть читает фразу, заканчивающееся точкой, с консоли, и распечатывает предложение в его положительной форме. *Листинг 11-2* показывает демонстрационное консольное взаимодействие с OPTIMIST. OPTIMIST работает хорошо, если предложения простые, но сложные предложения путают программу.

В строке 13 задается словарь отрицательных слов OPTIMIST с соответствующими положительными словами в строке 15. Таким образом, never становится always, и none становится all. OPTIMIST заменяет слово not пустой строкой. Строки 17 - 20 объявляют прописные и строчные буквы для перевода регистра в секции обработки фразы.

OPTIMIST создает каждую последующую входную фразу между строками 28 и 32, где DO-группа читает другое слово и присоединяет слово в конце фразы. В заголовке DO WHILE SUBSTR проверяет точку в конце.

**Примечание**: OPTIMIST может принять предложение, максимальная длина которого только 254 символа. PL/I отбрасывает любые дополнительные символы.

После чтения всей фразы OPTIMIST преобразует все символы верхнего регистра в нижний регистр для сканирования негативных слов. Она выполняет это преобразование регистра в строке 33 при помощи встроенной функции TRANSLATE. Программа OPTIMIST использует встроенную функцию VERIFY в строке 34, чтобы гарантировать, что предложение состоит только из букв и точки.

Если фраза содержит символы отличные от букв или точки, функция VERIFY возвращает первую ненулевую позицию, которая не соответствует, и OPTIMIST отвечает:

**Actually, that's an interesting idea.** (На самом деле, это интересная идея.)

Если функция VERIFY возвращает нулевое значение, то предложение содержит только переведенные строчные буквы и точку. В этом случае управление передается DO-группе между строками 36 и 42. На каждой итерации OPTIMIST использует встроенную функцию INDEX для поиска следующего отрицательного слова, заданного отрицательным (i). Если найдено, она устанавливает j в позицию отрицательного слова, и в операторе присваивания в строке 39, заменяет его соответствующим положительным словом. В этом присвоении часть фразы, которая встречается перед отрицательным словом определяется

**substr(sent,l,j-1)**

в то время как значение для замены отрицательного слова определяется,

**positive(i)**

и часть фразы, следующая за заменяемым отрицательным словом определяется по формуле:

**substr(sent,j+length(negative(i)))**

OPTIMIST объединяет эти три сегмента, чтобы произвести новое предложение в котором отрицательное слово заменено положительным. Затем она отправляет получившуюся фразу на консоль и возвращается назад, чтобы считать следующий ввод. Поскольку у всех отрицательных слов имеется начальный пробел, отрицательная часть всегда находится в начале слова. Таким образом OPTIMIST производит замену по умному. Это может привести к интересным результатам.

Вы можете сделать по крайней мере три улучшения программы OPTIMIST. Во-первых, если предложение превышает 254 символа, входное сканирование никогда не останавливается, потому что точка не найдена. Вы можете включать проверку, чтобы гарантировать, что недавно добавленное слово не превышает максимальный размер.

Во-вторых, в DO-группе между строками 25 и 45 обработка состояния отсутствует, таким образом, OPTIMIST никогда не прекращает говорить. Она завершается только посредством ввода конца файла Ctrl-Z или Ctrl–C - системного "теплого" запуска. Вы можете включать ON блок, чтобы обнаружить конец файла для завершения программы разумным способом.

Наконец, вы можете попытаться сделать OPTIMIST более умной!

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа демонстрирует обработку символьной \*/**

**3 a /\* строки PL/I, превращая отрицательную фразу в \*/**

**4 a /\* положительную. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── optimist:**

**7 b │ procedure options(main);**

**8 b │ %replace**

**9 b │ true by '1'b,**

**10 b │ false by '0'b,**

**11 b │ nwords by 5;**

**12 b │ declare**

**13 b │ negative (1:nwords) character(8) varying static initial**

**14 b │ (' never',' none',' nothing',' not',' no'),**

**15 b │ positive (1:nwords) character(10) varying static initial**

**16 b │ (' always',' all',' something','',' some'),**

**17 b │ upper character(28) static initial**

**18 b │ ('ABCDEFGHIJKLMNOPQRSTUVWXYZ. '),**

**19 b │ lower character(28) static initial**

**20 b │ ('abcdefghijklmnopqrstuvwxyz. '),**

**21 b │ sent character(254) varying,**

**22 b │ word character(32) varying,**

**23 b │ (i,j) fixed;**

**24 b │**

Листинг 11‑1 Программа OPTIMIST

**25 c │ ┌── do while(true);**

**26 c │ │ put skip list('What''s up? ');**

**27 c │ │ sent = ' ';**

**28 d │ │ ┌── do while**

**29 d │ │ │ (substr(sent,length(sent)) ^= '.');**

**30 d │ │ │ get list (word);**

**31 d │ │ │ sent = sent !! ' ' !! word;**

**32 d │ │ └── end;**

**33 c │ │ sent = translate(sent,lower,upper);**

**34 c │ │ if verify(sent,lower) ^= 0 then**

**35 c │ │ sent = ' that''s an interesting idea.';**

**36 d │ │ ┌── do i = 1 to nwords;**

**37 d │ │ │ j = index(sent,negative(i));**

**38 d │ │ │ if j ^= 0 then**

**39 d │ │ │ sent = substr(sent,1,j-1) !!**

**40 d │ │ │ positive(i) !!**

**41 d │ │ │ substr(sent,j+length(negative(i)));**

**42 d │ │ └── end;**

**43 c │ │ put list('Actually,'!!sent);**

**44 c │ │ put skip;**

**45 c │ └── end;**

**46 b │**

**47 b └── end optimist;**

Листинг 11-1 Программа OPTIMIST (продолжение)

A>**optimist**

What's up? **Nothing is up.**

Actually, something is up.

What's up? **This is not fun.**

Actually, this is fun.

What's up? **Programs like this never make sense.**

Actually, programs like this always make sense.

What's up? **Nothing is easy that is not complicated.**

Actually, something is easy that is complicated.

What's up? **Nobody cares and its none of your business.**

Actually, somebody cares and its all of your business.

What's up? **The price of everything.**

Actually, the price of everything.

What's up? **Boy are you stupid.**

Actually, boy are you stupid.

What's up? **Dont get smart with me.**

Actually, dont get smart with me.

Листинг 11‑2 Взаимодействие с программой OPTIMIST

What's up? **You started it I didnt.**

Actually, you started it i didnt.

What's up? **No I did not.**

Actually, some i did.

What's up? **Thats better.**

Actually, thats better.

What's up? **You are hard to talk to.**

Actually, you are hard to talk to.

What's up? **There you go again.**

Actually, there you go again.

What's up? **Thats it I quit.**

Actually, thats it i quit.

What's up? **Stop that.**

Actually, stop that.

What's up? **If you dont stop I will pull your plug.**

Actually, if you dont stop i will pull your plug.

What's up? **You can not pull my plug.**

Actually, you can pull my plug.

What's up? **I know.**

Actually, i know.

What's up? **^Z**

END OF FILE (1), File: SYSIN=CON

Traceback: 09C5 0970 0157 4100 # 0909 0529 8090 0157

A>

Листинг 11-2 Взаимодействие с программой OPTIMIST (продолжение)

## Функция лексического анализа

Этот раздел представляет более практическое применение обработки строк. Часто полезно иметь в программе отдельную подпрограмму, которая читает строку ввода и разделяет ее на отдельные числа и символы. Такую подпрограмму называют лексическим анализатором или сканером свободного поля. Программа FSCAN, показанная в *Листинге 11-3*, является примером лексического анализатора.

FSCAN демонстрирует встроенную подпрограмму под названием GNT (Get Next Token), которая разбирает входную строку на отдельные элементы, называемые лексемами. Как только вы протестируете GNT, вы можете извлечь ее из этой программы и поместить ее в рабочую программу при необходимости. В [*Разделе 13.4*](#_Механизм_вычисления_арифметических) GNT используется при вычислении значений арифметических выражений.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа тестирует процедуру GNT, сканер свобод- \*/**

**3 a /\* ного поля (синтаксический анализатор), которая читает \*/**

**4 a /\* входную строку и разделяет ее на отдельные части. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── fscan:**

**7 b │ procedure options(main);**

**8 b │ %replace**

**9 b │ true by '1'b;**

**10 b │ declare**

**11 b │ token character(80) varying**

**12 b │ static initial('');**

**13 b │**

**14 b │ ┌── gnt:**

**15 c │ │ procedure;**

**16 c │ │ declare**

**17 c │ │ i fixed,**

**18 c │ │ line character(80) varying**

**19 c │ │ static initial('');**

**20 c │ │**

**21 c │ │ line = substr(line,length(token)+1);**

**22 d │ │ ┌── do while(true);**

**23 d │ │ │ if line = '' then**

**24 d │ │ │ get edit(line) (a);**

**25 d │ │ │ i = verify(line,' ');**

**26 d │ │ │ if i = 0 then**

**27 d │ │ │ line = '';**

**28 d │ │ │ else**

**29 e │ │ │ ┌── do;**

**30 e │ │ │ │ line = substr(line,i);**

**31 e │ │ │ │ i = verify(line,'0123456789.');**

**32 e │ │ │ │ ┌── if i = 0 then**

**33 e │ │ │ │ │ token = line;**

**34 e │ │ │ │ └── else**

**35 e │ │ │ │ ┌── if i = 1 then**

**36 e │ │ │ │ │ token = substr(line,1,1);**

**37 e │ │ │ │ └── else**

**38 e │ │ │ │ token = substr(line,1,i-1);**

**39 e │ │ │ │ return;**

**40 e │ │ │ └── end;**

**41 d │ │ └── end;**

**42 c │ └── end gnt;**

**43 b │**

**44 c │ ┌── do while(true);**

**45 c │ │ call gnt;**

**46 c │ │ put edit(''''!!token!!'''') (x(1),a);**

**47 c │ └── end;**

**48 b │**

**49 b └── end fscan;**

Листинг 11‑3 Программа FSCAN

*Листинг 11-4* показывает взаимодействие с программой FSCAN. FSCAN читает входную строку, разбирает строку на отдельные лексемы, и затем записывает лексемы заключенные в апострофы обратно в консоль. Лексемы - просто числовые значения, такие как 1234.56, или отдельные буквы и специальные символы. При сканировании лексем GNT пропускает все предшествующие пробелы между лексемами.

Программа FSCAN состоит из трех частей. Первая часть, строки 10 - 12, определяет область глобальных данных с именем token, используемую процедурой GNT. Вторая часть, строки 14 - 42, является самой процедурой GNT. Третья часть DO-группа между строками 44 и 47, которая выполняет тест процедуры функции GNT.

A>**fscan**

**88+9.9**

'88' '+' '9.9'

**1234567 89.10**

'1234567' '89.10'

**1,2,3,4,5,6,7**

'1' ',' '2' ',' '3' ',' '4' ',' '5' ',' '6' ',' '7'

**.... 666 .... 7.7.7.**

'....' '666' '....' '7.7.7.'

**^Z**

End of File (7), File: SYSIN=CON

Traceback: 08A1 23D1 0143 00FF # 08AB 06B9 0143 01F5

A>

Листинг 11‑4 Взаимодействие с программой FSCAN

### Процедура GNT

GNT хранит входную строку в символьной переменной с именем line, которая первоначально пуста из-за объявления в строке 18. При первом вызове GNT извлекает первую часть строки и помещает ее в token, которая становится следующим входным элементом. При каждом последующем вызове GNT удаляет предыдущее значение token с начала строки прежде, чем сканировать следующий элемент.

Например, предположим, что входная строка

⌴⌴⌴**88\*9.9**

где ⌴ представляет знак пробела. При первом вызове GNT token и line являются пустыми строками. Присвоение в строке 21 удаляет предыдущее значение token и оставляет line как пустую строку. DO-группа между строками 22 и 41 гарантирует, что буфер строки всегда заполнен. Если GNT встречается с пустым буфером, оператор GET EDIT в строке 24, сразу пополняет его. Вызов встроенной функции VERIFY в строке 25 возвращает первую позицию в строке, которая не является пробелом.

Если VERIFY возвращает 0, то вся строка пустая и должна быть очищена. Операция пополнения происходит на следующей итерации. Если строка - не полностью пустая, то управление передается в DO группу, начинающуюся в строке 29.

### DO-группы

Обработка в DO-группе происходит следующим образом. На входе значении i - первая непустая позиция буфера строки. Таким образом оператор в строке 30 удаляет предшествующие пробелы из строки, так что, следующая лексема начинается в первой позиции. Затем GNT вызывает функцию VERIFY, чтобы определить, является ли следующий элемент в строке числом.

Оператор присваивания в строке 31 устанавливает i равным 0, если весь буфер состоит из чисел и десятичных точек. В строке 31 i устанавливается равным 1, если первый элемент не является числом или точкой. Он присваивает i значение большее чем 1, если первый элемент является числом, которое не продолжается через весь буфер строки. Таким образом эта последовательность тестов, начинающихся в строке 32, или извлекает всю строку (i=0), первый символ строки (i=1), или первую часть строки (i>1).

В предыдущем примере входной строки, на первой итерации GNT устанавливает строку

**⌴ ⌴ ⌴ 8 8 \* 9 . 9**

1 2 3 4 5 6 7 8 9

где индексы в строке от 1 до 9, показаны ниже каждого символа. В строке 30 GNT удаляет начальные пробелы, оставляя строку в виде:

**8 8 \* 9 . 9**

1 2 3 4 5 6

В строка 31 вызывается функцию VERIFY, которая определяет местоположение первой позиции, содержащей нецифровой символ или точку. В данном случае VERIFY возвращает значение 3, которое соответствует \* в позиции 3. В результате тестов FSCAN выполняет строку 38 и производит эквивалент:

**substr('88\*9.9',1,2)**

Это приводит к значению token 88, которое является следующим числом в строке.

При следующем вызове GNT удаляет лексему из строки, используя операцию SUBSTR в строке 21 и оставляет строку в виде:

**\* 9 . 9**

1 2 3 4

Функция VERIFY в строке 31 возвращает значение 1, так как ведущая позиция line не является цифрой или точкой. Строка 36 извлекает и присваивает первый символ line в качестве значения token.

Третий вызов GNT получает последнюю лексему в строке сначала извлекая первый символ строки. Он оставляет строку в виде:

**9 . 9**

1 2 3

В этот раз, так как все символы являются цифрами или точками, функция VERIFY возвращает 0, и GNT выполняет строку 33. Это приводит к значению token 9.9, которое является остатком строки.

Четвертый вызов GNT очищает предыдущее значение лексемы из строки, чтобы строка была пустой строкой. Это заставляет GNT выполнять оператор GET EDIT в строке 24 и заполнить line с консоли. FSCAN продолжает выполнение аналогичным образом, пока вы не останавливаете его с помощью ввода Ctrl-C или Ctrl-Z.

Этот простой синтаксический анализатор имеет некоторые очевидные дефекты. Он не перехватывает состояние конца файла. Вы можете включать ON-блок, чтобы обнаружить это состояние, и возвратить нулевое значение token, чтобы обозначить отсутствие ввода. Кроме того, GNT не обнаруживает несколько символов точек. Что может вызвать в дальнейшем состояние преобразования (ERROR(1)) при попытке преобразования в десятичное значение.

Эти изменения дадут вам улучшенную версию GNT, которую вы сможете включить в любую из ваших программ.

Ссылки: Справочное руководство разделы 3.2, 6.4, 6.8, 13.7

**Конец раздела 10**

# Обработка списков

Для некоторых программ трудно определить точные требования к памяти перед запуском программы. Обработка списка - пример такой программы, поскольку число элементов данных может значительно изменяться во время работы программы.

PL/I имеет подпрограммы в библиотеке подпрограмм времени выполнения, которые динамично управляют выделением памяти. Когда операционная система загружает программу PL/I в область или раздел транзитных программ (TPA), PL/I сначала инициализирует всю оставшуюся свободную память в виде связного списка. Элементы списка содержат информационные поля и указатели на другие элементы списка. Программа динамично выделяет память при помощи оператора ALLOCATE и освобождает память, используя оператор FREE. PL/I постоянно сохраняет все сегменты памяти соединенными друг с другом при помощи связного списка.

Программы в этом разделе иллюстрируют обработку списков в двух случаях, в которых не просто предопределить объем требуемого хранения.

## Базированные переменные и указатели

Вы можете представить себе базированную переменную как шаблон, который соответствует области памяти, но не имеет хранения, выделенного непосредственно ей. Переменная указателя - просто двухбайтовое значение, которое содержит адрес переменной. Когда вы используете переменную указателя, вы программно помещаете этот шаблон базированной переменной в определенную часть памяти. Способ зависит от формы объявления базированной переменной.

Если объявление базированной переменной не содержит подразумеваемую основу, то вы должны определить любую ссылку на базированную переменную с помощью указателя. Если объявление базируемой переменной включает подразумеваемую основу, то вы можете включать спецификатор указателя в любую ссылку на базированную переменную, или просто использовать подразумеваемый указатель, заданный в объявлении в качестве основы.

Рассмотрим следующее объявление в качестве примера:

**declare  
 i fixed,  
 mat(0:5) fixed,  
 (p, q) pointer,  
 x fixed based,  
 y fixed based(p),  
 z fixed based(f());**

PL/I выделяет память под две переменные i и mat, потому что они не базированные переменные. PL/I также присваивает места хранения для двух переменных указателя p и q. Однако три переменные x, y и z объявлены как базированные переменные, и у них нет мест хранения до выполнения. Вместо этого, PL/I определяет их фактические адреса хранения в процессе выполнения программы. Переменная x не имеет подразумеваемой основы, таким образом, каждая ссылка на x должна иметь спецификатор указателя, такой как:

**p->x = 5;**

или

**q->x = 6;**

Первый оператор присваивает значение 5 двухбайтовой переменной FIXED BINARY в ячейке памяти, заданной p. Второй присваивает значение 6 расположению, заданному q.

С другой стороны, переменная y, имеет подразумеваемую основу, и вы можете сослаться на нее с или без спецификатора указателя. Ссылка

**y = 5;**

Эквивалентна

**p->y = 5;**

и таким образом,

**y = 5;** и **q->y = 6;**

имеют точно такой же эффект, как два предыдущих присвоения x.

Переменная z, как и переменная y имеет подразумеваемую основу. В этом случае основой является ссылка на указатель-функции без аргументов. Например, функция f может принять форму:

**f:  
 procedure returns(pointer);  
 return (addr(mat(i)));  
end f;**

Используя это определение f, вы можете сослаться на z следующим образом:

**p->z = 5;**

или

**z = 6;**

Первая форма эквивалентна показанным выше, с расположением, полученным из переменной указателя p. Однако, вторая форма является сокращением для:

**f() -> z = 6;**

В этом случае PL/I вычисляет функцию f для получения адреса базированной переменной z. Эта форма имеет двойное преимущество. Во-первых, выражение указателя может быть сложным, и не ограничивается простой переменной указателя. Во-вторых, код для функции f появляется только один раз, вместо того, чтобы быть дублированным в каждой ссылке переменной. Это может сэкономить значительное место в программе.

**Примечание**: подразумеваемая основа должна быть в пределах объявления базированной переменной.

Следующая неправильная последовательность кода иллюстрирует это понятие:

**┌── main:  
│ procedure options(main);  
│ declare  
│ x based(p),  
│ y based(q),  
│ p pointer;  
│ ┌── begin;  
│ │ declare  
│ │ (p,q) pointer;  
│ │ x = 5;  
│ │ y = 10;  
│ └── end;  
│ declare  
│ q pointer;  
└── end main;**

Поскольку переменные x и y основываются на p и q, указатели p и q должны быть в тот же или охватывающий области. Здесь указатели p и q объявлены во встроенном блоке BEGIN, который является другой средой.

## Программа REVERSE

Нашим первым примером обработки списков является программа под названием REVERSE. Программа OPTIMIST в [*Разделе 11.1*](#_Программа_OPTIMIST) может принять предложение максимум из 254 символов с максимальной длиной строки. Однако, REVERSE принимает предложение фактически любой длины при помощи структуры списка, вместо единственной символьной строки. Вместо того, чтобы выполнить замену слов, REVERSE просто меняет входное предложение.

*Листинг 12-1* показывает программу REVERSE, которая разделена на три части. Первая часть, строки 12-17, читает предложение с консоли и записывает предложение обратно в консоль в обратном порядке. Каждое входное предложение состоит из последовательности слов до 35 символов в длину. Это достаточно, чтобы содержать

**supercalifragilisticexpialidocious[[5]](#footnote-5)**

одно из самых длинных слов английского языка.

Для упрощения обработки ввода, REVERSE требует пробел перед точкой, которая завершает предложение. REVERSE также заканчивает выполнение, когда вы вводите пустое предложение.

Вторая часть REVERSE - отдельная подпрограмма, названная read\_it, которая начинается в строке 19. Третья часть - подпрограмма названная write\_it, которая начинается в строке 37. Создание этих функций отдельными подпрограммами в главной программе упрощает общую структуру.

*Листинг 12-2* показывает консольное взаимодействие с REVERSE.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа читает предложение и инвертирует его.\*/**

**3 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**4 a ┌── reverse:**

**5 b │ procedure options(main);**

**6 b │ declare**

**7 b │ sentence pointer,**

**8 b │ 1 wordnode based (sentence),**

**9 b │ 2 word character(30) varying,**

**10 b │ 2 next pointer;**

**11 b │**

**12 c │ ┌── do while('1'b);**

**13 c │ │ call read\_it();**

**14 c │ │ if sentence = null then**

**15 c │ │ stop;**

**16 c │ │ call write\_it();**

**17 c │ └── end;**

Листинг 12‑1 Программа REVERSE

**18 b │**

**19 b │ ┌── read\_it:**

**20 c │ │ procedure;**

**21 c │ │ declare**

**22 c │ │ newword character(30) varying,**

**23 c │ │ newnode pointer;**

**24 c │ │ sentence = null;**

**25 c │ │ put skip list('What''s up? ');**

**26 d │ │ ┌── do while('1'b);**

**27 d │ │ │ get list(newword);**

**28 d │ │ │ if newword = '.' then**

**29 d │ │ │ return;**

**30 d │ │ │ allocate wordnode set (newnode);**

**31 d │ │ │ newnode->next = sentence;**

**32 d │ │ │ sentence = newnode;**

**33 d │ │ │ word = newword;**

**34 d │ │ └── end;**

**35 c │ └── end read\_it;**

**36 b │**

**37 b │ ┌── write\_it:**

**38 c │ │ procedure;**

**39 c │ │ declare**

**40 c │ │ p pointer;**

**41 c │ │ put skip list('Actually, ');**

**42 d │ │ ┌── do while (sentence ^= null);**

**43 d │ │ │ put list(word);**

**44 d │ │ │ p = sentence;**

**45 d │ │ │ sentence = next;**

**46 d │ │ │ free p->wordnode;**

**47 d │ │ └── end;**

**48 c │ │ put list('.');**

**49 c │ │ put skip;**

**50 c │ └── end write\_it;**

**51 b │**

**52 b └── end reverse;**

Листинг 12-1 Программа REVERSE (продолжение)

A>**reverse**

What's up? **North is up**

Actually, up is North

What's up? **The rain in Spain falls mainly in the plain**

Actually, plain the in mainly falls Spain in rain The

What's up? **3 + 5 = 8**

Actually, 8 = 5 + 3

What's up?

A>

Листинг 12‑2 Взаимодействие с программой REVERSE

Программа REVERSE хранит каждое слово в отдельной области памяти, полученной с помощью оператора ALLOCATE в строке 30. На каждой итерации DO-группы оператор ALLOCATE получает уникальный участок свободной памяти, достаточно большой, чтобы содержать структуру wordnode, определенную в строке 8. Элементы wordnode соединены через поле next каждого выделения, и начало списка определяется значением переменной указателя sentence.

Каждое выделение использует 38 байтов. Вы можете проверить это, исследовав таблицу символов. Структура wordnode 38 байтов длиной, потому что word объявлено как CHARACTER(35) VARYING, и требует один байт для хранения текущей длины, 35 байтов для значения самой строки, и сопровождается двухбайтовым значением указателя.

Например, если входное предложение

**I SHALL RETURN .**

REVERSE выполняет оператор ALLOCATE три раза, один раз для каждого слова в списке.

Предположим, что эти три выделения памяти расположены в адресах 1000, 2000 и 3000. Программа REVERSE начинает чтение sentence в основной DO-группе в процедуре read\_it. Она инициализирует указатель sentence нулевым адресом (0000). После ввода в DO-группе начинающейся в строке 26, значение sentence выглядит следующим образом:

**SENTENCE: 0000**

REVERSE читает первое слово с помощью оператора GET в строке 27, и потому что значение не является точкой, она выделяет первую 38-байтовую область, чтобы содержать слово. Поскольку она составляет предложение, REVERSE помещает значение указателя переменной sentence в поле next и введенное слово в поле word. Затем прочитанное предыдущее слово становится новым заголовком списка. После обработки слова I список выглядит следующим образом:

**SENTENCE: 1000**

┌──────┐

**1000:**│ I │

├──────┤

│ 0000 │

└──────┘

Затем REVERSE снова обрабатывает цикл. На этот раз она читает, слово SHALL и выделят вторую 38-байтовую область. Новая выделенная область становится новым заголовком списка со следующей структурой указателя:

**SENTENCE: 2000**

┌───────┐┌──────┐

**2000:**│ SHALL │ ┌─> **1000:**│ I │

├───────┤ │ ├──────┤

│ 1000 ├─┘ │ 0000 │

└───────┘ └──────┘

REVERSE повторяет цикл еще раз и обрабатывает последнее слово - RETURN и выделяет заключительную 38-байтовую область, помещая ее во главе списка, в результате возникает следующая последовательность узлов:

**SENTENCE: 3000**

┌────────┐┌───────┐┌──────┐

**3000:**│ RETURN │ ┌─> **2000:**│ SHALL │ ┌─> **1000:**│ I │

├────────┤ │ ├───────┤ │ ├──────┤

│ 1000 ├─┘ │ 1000 ├─┘ │ 0000 │

└────────┘ └───────┘ └──────┘

Программа следует за указателем переменной sentence в структуре к узлу с RETURN, затем к узлу с SHALL, и наконец к узлу с I, где она встречается с значением конца списка - 0000.

На самом деле REVERSE создает список в обратном порядке. DO-группа в процедуре **write\_it**, строки 42 - 47, просто осуществляет поиск в списке, начиная с указателя sentence, и распечатывает каждое слово, с которым она встречается. Как только слово записано, оператор FREE в строке 46 освобождает 38-байтовую область, выделенную ему. Процедура **write\_it** перемещает указатель переменной sentence к следующему элементу в списке перед выполнением оператора FREE, чтобы освободить текущий элемент.

**Примечание**: Хранение не остается неповрежденным после освобождения.

Преимущество списка структур состоит в том, что предложение может быть произвольно длинным, ограниченным только размером доступной памяти. Недостатком, конечно, является значительно больший объем памяти занимаемый предложениями, которые могут быть представлены строкой из 254 символов.

## Программа сетевого планирования

Следующий пример обширен и иллюстрирует два тезиса. Во-первых, он демонстрирует возможности конструкций PL/I, обрабатывающих списки. Во-вторых, он показывает, как разделить большую, сложную программу на маленькие, отдельные логически модули, и таким образом упростить задачу кодирования.

Программа NETWORK, показанная в *Листинге 12-4*, выполняет сетевое планирование. Т.е. она находит кратчайший путь между узлами в сети. Пользователь вводит сеть городов и расстояний между городами. Затем NETWORK создает связанный набор узлов, используя обработку списка структур. По требованию пользователя NETWORK вычисляет кратчайший путь из всех городов в сети до заданного места назначения, и затем выборочно отображает конкретные оптимальные пути через сеть.

Работу программы проще понять, если вы сначала исследуете консольное взаимодействие, показанное в *Листинге 12-3*. Во-первых, вы вводите список городов и расстояний между городами, заканчивая ввод Ctrl-Z. Ввод Ctrl-Z вызывает отображение всей сети, чтобы помочь в обнаружении ошибок ввода. Затем NETWORK предлагает вам определить город назначения, в этом случае, Tijuana, и начальный город, в этом случае, Boise.

Затем NETWORK отображает оптимальный маршрут. Может быть несколько равной длины. Далее, NETWORK запрашивает другой начальный город. Если вы вводите Ctrl-Z. NETWORK возвращается к другой целевой подсказке, оставляя сеть неповрежденной. Взаимодействие продолжается этим способом, пока вы не вводите Ctrl-Z в ответ на подсказку назначение. Когда это происходит, NETWORK очищает сеть и возвращается, чтобы принять совершенно новую сеть городов и расстояний. Вся программа заканчивается, если вы вводите пустую сеть в этот момент, например, Ctrl-Z.

A>**network**

Type "City1, Dist, City2"

**Seattle, 150, Boise**

**Boise, 300, Modesto**

**Seattle, 400, Modesto**

**Modesto, 150, Monterey**

**Modesto, 50, San-Francisco**

**San-Francisco, 200, Las-Vegas**

Листинг 12‑3 Взаимодействие с программой NETWORK

**Las-Vegas, 350, Monterey**

**Los-Angeles, 400, Las-Vegas**

**Bakersfield, 300, Monterey**

**Bakersfield, 250, Las-Vegas**

**Los-Angeles, 450, Tijuana**

**Tijuana, 700, Las-Vegas**

**Las-Vegas, 920, Boise**

**Pacific-Grove, 5, Monterey**

**^Z**

Pacific-Grove :

5 miles to Monterey

Tijuana :

700 miles to Las-Vegas

450 miles to Los-Angeles

Bakersfield :

250 miles to Las-Vegas

300 miles to Monterey

Los-Angeles :

450 miles to Tijuana

400 miles to Las-Vegas

Las-Vegas :

920 miles to Boise

700 miles to Tijuana

250 miles to Bakersfield

400 miles to Los-Angeles

350 miles to Monterey

200 miles to San-Francisco

San-Francisco :

200 miles to Las-Vegas

50 miles to Modesto

Monterey :

5 miles to Pacific-Grove

300 miles to Bakersfield

350 miles to Las-Vegas

150 miles to Modesto

Modesto :

50 miles to San-Francisco

150 miles to Monterey

400 miles to Seattle

300 miles to Boise

Boise :

920 miles to Las-Vegas

300 miles to Modesto

150 miles to Seattle

Seattle :

400 miles to Modesto

150 miles to Boise

Листинг 12-3 Взаимодействие с программой NETWORK (продолжение)

Type Destination **Tijuana**

Type Start **Boise**

1250 miles remain, 300 miles to Modesto

950 miles remain, 50 miles to San-Francisco

900 miles remain, 200 miles to Las-Vegas

700 miles remain, 700 miles to Tijuana

Type Start **^Z**

Type Destination **Pacific-Grove**

Type Start **Seattle**

555 miles remain, 400 miles to Modesto

155 miles remain, 150 miles to Monterey

5 miles remain, 5 miles to Pacific-Grove

Type Start **^Z**

Type Destination **^Z**

Type "City1, Dist, City2"

**^Z**

A>

Листинг 12-3 Взаимодействие с программой NETWORK (продолжение)

### Список структур NETWORK

NETWORK использует две структуры данных в качестве элементов списка. Первая структура называется city\_node и соответствует определенному городу. Она определена в строке 16 *Листинга 12-4*. Следующий пример показывает структуру city\_node:

**┌────────────────┐  
CITY\_NODE: │ city\_name │  
 ├────────────────┤  
 │ total\_distance │  
 ├────────────────┤  
 │ investigate │  
 ├────────────────┤  
 │ city\_list │  
 ├────────────────┤  
 │ route\_head │  
 └────────────────┘**

Поле city\_name содержит значение символьной строки названия города, в то время как поля total\_distance и investigate используются процедурой shortest-distance. Значения указателей city\_list и route\_head соединяют все города в сети.

Вторая структура называется route\_node и определена в строке 23. route\_node устанавливает единственное соединение между городом и одним из его соседей. Вы выделяете несколько route\_node для города в соответствии с количеством соединений его с соседними городами. Структура route\_node показана ниже:

**┌────────────────┐  
ROUTE\_NODE:│ next\_city │  
 ├────────────────┤  
 │ route\_distance │  
 ├────────────────┤  
 │ route\_list │  
 └────────────────┘**

Список route\_node, связанных с конкретным городом, начинается со значения указателя, названного route\_head, который является частью структуры city\_node. Маршрут определяет следующий указатель route\_list на дополнительные route\_node, пока вы не встретите route\_node с нулевой записью в route\_list. Каждый route\_node также имеет значение указателя, обозначенного next\_city, который приводит к соседнему city\_node, вместе с полем route\_distance, которое определяет расстояние в милях до следующего города.

Следующий пример иллюстрирует это понятие. Предположим, что Monterey в 350 милях от Las Vegas. NETWORK должна выделить два city\_node и два route\_node с демонстрационными адресами слева от каждого выделения следующим образом. Вы можете временно проигнорировать поля отмеченные в схеме x.

**CITY\_NODE CITY\_NODE  
 ┌────────────────┐ ┌────────────────┐  
 1000 │ Monterey │ 2000 │ Las Vegas │  
 ├────────────────┤ ├────────────────┤  
 │ xxxxxxxx │ │ xxxxxxxx │  
 ├────────────────┤ ├────────────────┤  
 │ xxxxxxxx │ │ xxxxxxxx │  
 ├────────────────┤ ├────────────────┤  
 │ xxxxxxxx │ │ xxxxxxxx │  
 ├────────────────┤ ├────────────────┤  
 │ 3000 │ │ 4000 │  
 └────────────────┘ └────────────────┘  
  
 ROUTE\_NODE ROUTE\_NODE  
 ┌────────────────┐ ┌────────────────┐  
 3000 │ 1000 │ 4000 │ 1000 │  
 ├────────────────┤ ├────────────────┤  
 │ 350 │ │ 350 │  
 └────────────────┘ └────────────────┘**

Связанный список, начинающийся в city\_head, приводит ко всем городам в сети. Учитывая два предыдущих города, список городов выглядит следующим образом:

**CITY HEAD  
 ┌────────────────┐  
 │ 1000 │  
 └────────────────┘  
  
 CITY\_NODE CITY\_NODE  
 ┌────────────────┐ ┌────────────────┐  
 1000 │ Monterey │ ┌─> 2000 │ Las Vegas │  
 ├────────────────┤ │ ├────────────────┤  
 │ xxxxxxxx │ │ │ xxxxxxxx │  
 ├────────────────┤ │ ├────────────────┤  
 │ xxxxxxxx │ │ │ xxxxxxxx │  
 ├────────────────┤ │ ├────────────────┤  
 │ 2000** ├──┘ **│ 0000 │  
 ├────────────────┤ ├────────────────┤  
 │ xxxxxxxx │ │ xxxxxxxx │  
 └────────────────┘ └────────────────┘**

### Обход связанных списков

Некоторые из процедур в NETWORK используют одну конкретную форму итеративной DO-группы, чтобы обойти связанные списки. Оператор в строке 95 типичен:

**do p = city\_head repeat (p->city\_list) while (p^=null);**

Заголовок DO-группы последовательно обрабатывает каждый элемент связанного списка, начинающегося в city\_head, пока он не встретится с нулевой ссылкой 0000. На первой итерации DO-группа устанавливает переменную указателя p в значение переменной указателя city\_head. В примере показанном выше, это приводит к присвоению p = 1000.

На следующей итерации p берет значение поля city\_list - 1000, которое адресует Las Vegas. Это приводит к значению p = 2000. На последней итерации p берет значение поля city\_list, базируемого в 2000, приводя к p = 0000. После чего DO-группа прекращает выполняться, так как p равен нулю.

### Общая структура программы

С учетом вышеизложенного, рассмотрим общую структуру программы. Программные вызовы верхнего уровня происходят в DO-группе между строками 31 и 38. Остальная часть программы полностью состоит из вложенных подпрограмм.

NETWORK логически разделена на четыре части:

* Секция ввода создает и поддерживает сеть городов, состоя из четырех процедур, начинающихся в строке 45: setup, connect, find и print\_all.
* Анализ кратчайшего пути между городами происходит в процедуре shortest\_distance, начинающейся в строке 164.
* Операции отображения кратчайшего пути разделены между двумя процедурами print\_paths и print\_route, соответственно.
* Процедура free\_all очищает старую сеть прежде, чем загрузить новую.

Начинаясь в строке 32, главная программа вызывает setup, чтобы считать сеть. Если city\_list пуст, то NETWORK останавливается. Иначе она вызывает print\_all для отображения сети, и затем вызывает print\_paths для запроса и отображения самых коротких маршрутов. По возврату, NETWORK вызывает free\_all, чтобы освободить хранение. Этот процесс продолжается, пока вы не вводите пустую сеть.

### Процедура setup (задание)

Основной цикл в setup расположен между строками 54 и 58. На каждой итерации оператор GET LIST в строке 55, читает пару городов с расстоянием между ними. Затем setup дважды вызывает подпрограмму connect, чтобы установить связь в обоих направлениях между городами. ON-блок в строке 50 перехватывает комбинацию клавиш Ctrl-Z.

### Процедура connect

Процедура connect организует единственный route\_node, чтобы соединить первый город со вторым городом. Процедура connect делает это, дважды вызывая процедуру find, один раз для первого города и один раз для второго города. Процедура find определяет местоположение города, если он существует в сети или создает city\_node, если он еще не существует. По возврату из find процедура connect создает и заполняет route\_node, строки 79 - 82.

В предыдущем примере первый вызов connect, устанавливает city\_node для Monterey и Las Vegas, косвенно через процедуру find, и затем производит route\_node только для Monterey. Второй вызов connect устанавливает route\_node для Las Vegas.

### Процедура find

Процедура find, начинающаяся в строке 89, ищет city\_list, начинающийся в city\_head, пока она не находит входной город или исчерпывает city\_list. Если входной город не существует, find создает его между строками 100 и 105. В любом случае find возвращает указатель на требуемый city\_node.

### Процедура print\_all

Процедура print\_all расположена между строками 113 и 127. NETWORK вызывает print\_all после создания сети. Эта процедура стартует c city\_head и отображает все города city\_list. Поскольку она посещает каждый город, print\_all также обходит и отображает route\_head. После завершения процедуры print\_all все city\_node и route\_node посещены и отображены.

### Процедура print\_paths

Процедура print\_paths читает город назначения в строке 143 и отправляет его в процедуру shortest\_distance. По возврату print\_paths устанавливает поле total\_distance каждого city\_node к total\_distance города назначения. Вы вводите стартовый город в строке 148, и print\_paths отправляет его в процедуру print\_route для выполнения отображения.

### Процедура print\_route

Процедура print\_route в строке 214 отображает оптимальный маршрут от входного города до места назначения. Процедура находит оптимальный маршрут следующим образом: общее расстояние от входного города до места назначения было уже вычислено и сохранено в поле total\_distance. Процедура получает первый участок наилучшего маршрута путем нахождения соседнего города, поле которого total\_distance отличается от точного расстояние до соседнего. Затем она отображает соседа, перемещается в соседний город и повторяет ту же операцию. В итоге, она достигает города назначения и завершает операцию отображения.

Строка 221 находит исходный city\_node. Строка 231 отображает остающееся расстояние, поиск первого или следующего участка происходит между строками 233 и 244. На каждой итерации, строка 236 проверяет, был ли сосед найден, чей total\_distance плюс расстояние участка соответствует текущему городу. Если соответствует, строка 238 отображает расстояние участка и поиск завершается установкой q значения null.

### Процедура shortest\_distance

Эта процедура берет входной город, называемый пунктом назначения, и вычисляет минимальное общее расстояние от каждого города в сети до пункта назначения. Затем она записывает это общее количество в каждом city\_node в поле total\_distance. При расчете минимального общего расстояния, процедура реализует следующий алгоритм:

1. Первоначально помечает все поля total\_distance значением infinite (бесконечность - 32767 в PL/I), чтобы указать, что узел в настоящее время не имеет связи.
2. Устанавливает флаг investigate в ложь для каждого города. Флаг investigate отмечает city\_node, которому требуется последующая обработка.
3. Устанавливает total\_distance до пункта назначения равным нулю, все остальные в настоящее время установлены равными бесконечности, но изменяются во время обработки.
4. Устанавливает флаг investigate в истину только для пункта назначения.
5. В city\_list исследуются city\_node, имеющие наименьшее значение total\_distance, и чей флаг investigate - истина. Сначала находится только место назначения. Когда ни у одного city\_node нет флага investigate, установленного в истину, обработка завершается и все поля с минимальными total\_distance вычислены.
6. Очищается флаг investigate для города, найденного в пункте 4, и извлекается текущее значение его поля total\_distance. Исследуется каждый из его соседей, если текущее поле total\_distance плюс расстояние участка - меньше, чем поле total\_distance, отмеченное у соседа, то поле total\_distance соседа заменяется этой суммой. Затем сосед отмечается для обработки, установкой флага investigate в истину. После обработки каждого соседа происходит возврат к пункту 4.

Таким образом, алгоритм следует по сети, разрабатывая кратчайший путь к любому узлу и, как следствие, посещает каждый город ровно один раз. Это вызвано тем, что процесс является линейным, и дополнительные узлы не влияют существенно на продолжительность анализа сети.

### Процедура free\_all

Заключительная процедура free\_all, начинающаяся в строке 251, возвращает сетевое хранение в конце обработки каждой сети. Процедура посещает и затем сбрасывает каждый city\_node и весь список соединений route\_node.

### Расширение программы NETWORK

Вы можете расширить NETWORK несколькими способами. Во-первых, можно открыть файл STREAM и считать граф с диска, потому что неудобно каждый раз вводить всю сеть при выполнении программы. Вы можете также сохранить несколько сетей на диске и получить их по команде с консоли.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа находит кратчайший путь между узлами \*/**

**3 a /\* в сети. Она содержит 8 внутренних процедур: \*/**

**4 a /\* SETUP, CONNECT, FIND, PRINT\_ALL, PRINT\_PATHS, \*/**

**5 a /\* SHORTEST\_DISTANCE, PRINT\_ROUTE и FREE\_ALL. \*/**

**6 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**7 a network:**

**8 b procedure options(main);**

**9 b %replace**

**10 b true by '1'b,**

**11 b false by '0'b,**

**12 b citysize by 20,**

**13 b infinite by 32767;**

**14 b declare**

**15 b sysin file;**

**16 b declare**

**17 b 1 city\_node based,**

**18 b 2 city\_name character(citysize) varying,**

**19 b 2 total\_distance fixed,**

**20 b 2 investigate bit,**

**21 b 2 city\_list pointer,**

**22 b 2 route\_head pointer;**

**23 b declare**

**24 b 1 route\_node based,**

**25 b 2 next\_city pointer,**

**26 b 2 route\_distance fixed,**

**27 b 2 route\_list pointer;**

**28 b declare**

**29 b city\_head pointer;**

**30 b**

**31 c ┌── do while(true);**

**32 c │ call setup();**

**33 c │ if city\_head = null then**

**34 c │ stop;**

**35 c │ call print\_all();**

**36 c │ call print\_paths();**

**37 c │ call free\_all();**

**38 c └── end;**

Листинг 12‑4 Программа NETWORK

**39 b**

**40 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**41 b /\* Эта процедура читает два города и затем вызывает \*/**

**42 b /\* процедуру CONNECT, чтобы установить соединение \*/**

**43 b /\* (в обоих направлениях) между городами. \*/**

**44 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**45 b ┌── setup:**

**46 c │ procedure;**

**47 c │ declare**

**48 c │ distance fixed,**

**49 c │ (city1, city2) character(citysize) varying;**

**50 c │ on endfile(sysin) goto eof;**

**51 c │ city\_head = null;**

**52 c │ put skip list('Type "City1, Dist, City2"');**

**53 c │ put skip;**

**54 d │ ┌── do while(true);**

**55 d │ │ get list(city1, distance, city2);**

**56 d │ │ call connect(city1, distance, city2);**

**57 d │ │ call connect(city2, distance, city1);**

**58 d │ └── end;**

**59 c │ eof:**

**60 c └── end setup;**

**61 b**

**62 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**63 b /\* Эта процедура устанавливает единственный route\_node,\*/**

**64 b /\* чтобы соединить первый город со вторым городом, \*/**

**65 b /\* вызывая процедуру FIND дважды, один раз для первого \*/**

**66 b /\* города и один раз для второго города. \*/**

**67 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**68 b ┌── connect:**

**69 c │ procedure(source\_city, distance, destination\_city);**

**70 c │ declare**

**71 c │ source\_city character(citysize) varying,**

**72 c │ destination\_city character(citysize) varying,**

**73 c │ distance fixed,**

**74 c │ (r, s, d) pointer;**

**75 c │**

**76 c │ s = find(source\_city);**

**77 c │ d = find(destination\_city);**

**78 c │ allocate route\_node set (r);**

**79 c │ r->route\_distance = distance;**

**80 c │ r->next\_city = d;**

**81 c │ r->route\_list = s->route\_head;**

**82 c │ s->route\_head = r;**

**83 c └── end connect;**

Листинг 12-4 Программа NETWORK (продолжение)

**84 b**

**85 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**86 b /\* Эта процедура выполняет поиск в списке городов и \*/**

**87 b /\* возвращает указатель на требуемый city\_node. \*/**

**88 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**89 b ┌── find:**

**90 c │ procedure(city) returns(pointer);**

**91 c │ declare**

**92 c │ city character(citysize) varying,**

**93 c │ (p, q) pointer;**

**94 c │**

**95 d │ ┌── do p = city\_head**

**96 d │ │ repeat(p->city\_list) while(p^=null);**

**97 d │ │ if city = p->city\_name then**

**98 d │ │ return(p);**

**99 d │ └── end;**

**100 c │ allocate city\_node set(p);**

**101 c │ p->city\_name = city;**

**102 c │ p->city\_list = city\_head;**

**103 c │ city\_head = p;**

**104 c │ p->total\_distance = infinite;**

**105 c │ p->route\_head = null;**

**106 c │ return(p);**

**107 c └── end find;**

**108 b**

**109 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**110 b /\* Эта процедура начинается в city\_head и отображает \*/**

**111 b /\* все города в city\_list. \*/**

**112 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**113 b ┌── print\_all:**

**114 c │ procedure;**

**115 c │ declare**

**116 c │ (p, q) pointer;**

**117 c │**

**118 d │ ┌── do p = city\_head**

**119 d │ │ repeat(p->city\_list) while(p^=null);**

**120 d │ │ put skip list(p->city\_name,':');**

**121 e │ │ ┌── do q = p->route\_head**

**122 e │ │ │ repeat(q->route\_list) while(q^=null);**

**123 e │ │ │ put skip list(q->route\_distance,'miles to',**

**124 e │ │ │ q->next\_city->city\_name);**

**125 e │ │ └── end;**

**126 d │ └── end;**

**127 c └── end print\_all;**

Листинг 12-4 Программа NETWORK (продолжение)

**128 b**

**129 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**130 b /\* Эта процедура читает город назначения, вызывает \*/**

**131 b /\* процедуру SHORTEST\_DISTANCE и устанавливает поле \*/**

**132 b /\* total\_distance в каждом city\_node равным общему \*/**

**133 b /\* расстоянию от города назначения. \*/**

**134 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**135 b ┌── print\_paths:**

**136 c │ procedure;**

**137 c │ declare**

**138 c │ city character(citysize) varying;**

**139 c │**

**140 c │ on endfile(sysin) goto eof;**

**141 d │ ┌── do while(true);**

**142 d │ │ put skip list('Type Destination ');**

**143 d │ │ get list(city);**

**144 d │ │ call shortest\_distance(city);**

**145 d │ │ on endfile(sysin) goto eol;**

**146 e │ │ ┌── do while(true);**

**147 e │ │ │ put skip list('Type Start ');**

**148 e │ │ │ get list(city);**

**149 e │ │ │ call print\_route(city);**

**150 e │ │ └── end;**

**151 d │ │ eol: revert endfile(sysin);**

**152 d │ └── end;**

**153 c │ eof:**

**154 c └── end print\_paths;**

**155 b**

Листинг 12-4 Программа NETWORK (продолжение)

**156 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**157 b /\* Эта процедура является сердцем программы. Она \*/**

**158 b /\* принимает входной город (назначения), и вычисляет \*/**

**159 b /\* минимальное общее расстояние от каждого города в \*/**

**160 b /\* сети до места назначения. Затем она записывает это \*/**

**161 b /\* минимальное значение в поле total\_distance каждого \*/**

**162 b /\* city\_node. \*/**

**163 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**164 b ┌── shortest\_distance:**

**165 c │ procedure(city);**

**166 c │ declare**

**167 c │ city character(citysize) varying;**

**168 c │ declare**

**169 c │ bestp pointer,**

**170 c │ (d, bestd) fixed,**

**171 c │ (p, q, r) pointer;**

**172 d │ ┌── do p = city\_head**

**173 d │ │ repeat(p->city\_list) while(p^=null);**

**174 d │ │ p->total\_distance = infinite;**

**175 d │ │ p->investigate = false;**

**176 d │ └── end;**

**177 c │ p = find(city);**

**178 c │ p->total\_distance = 0;**

**179 c │ p->investigate = true;**

**180 d │ ┌── do while(true);**

**181 d │ │ bestp = null;**

**182 d │ │ bestd = infinite;**

**183 e │ │ ┌── do p = city\_head**

**184 e │ │ │ repeat(p->city\_list) while(p^=null);**

**185 e │ │ │ if p->investigate then**

**186 f │ │ │ ┌── do;**

**187 f │ │ │ │ if p->total\_distance < bestd then**

**188 g │ │ │ │ ┌── do;**

**189 g │ │ │ │ │ bestd = p->total\_distance;**

**190 g │ │ │ │ │ bestp = p;**

**191 g │ │ │ │ └── end;**

**192 f │ │ │ └── end;**

**193 e │ │ └── end;**

**194 d │ │ if bestp = null then**

**195 d │ │ return;**

**196 d │ │ bestp->investigate = false;**

**197 e │ │ ┌── do q = bestp->route\_head**

**198 e │ │ │ repeat(q->route\_list) while(q^=null);**

**199 e │ │ │ r = q->next\_city;**

**200 e │ │ │ d = bestd + q->route\_distance;**

**201 e │ │ │ if d < r->total\_distance then**

**202 f │ │ │ ┌── do;**

**203 f │ │ │ │ r->total\_distance = d;**

**204 f │ │ │ │ r->investigate = true;**

**205 f │ │ │ └── end;**

**206 e │ │ └── end;**

**207 d │ └── end;**

**208 c └── end shortest\_distance;**

Листинг 12-4 Программа NETWORK (продолжение)

**209 b**

**210 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**211 b /\* Эта процедура отображает оптимальный маршрут от \*/**

**212 b /\* входного города до пункта назначения. \*/**

**213 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**214 b ┌── print\_route:**

**215 c │ procedure(city);**

**216 c │ declare**

**217 c │ city character(citysize) varying;**

**218 c │ declare**

**219 c │ (p,q) pointer,**

**220 c │ (t,d) fixed;**

**221 c │ p = find(city);**

**222 d │ ┌── do while(true);**

**223 d │ │ t = p->total\_distance;**

**224 d │ │ if t = infinite then**

**225 e │ │ ┌── do;**

**226 e │ │ │ put skip list('(No Connection)');**

**227 e │ │ │ return;**

**228 e │ │ └── end;**

**229 d │ │ if t = 0 then**

**230 d │ │ return;**

**231 d │ │ put skip list(t,'miles remain,');**

**232 d │ │ q = p->route\_head;**

**233 e │ │ ┌── do while(q^=null);**

**234 e │ │ │ p = q->next\_city;**

**235 e │ │ │ d = q->route\_distance;**

**236 e │ │ │ if t = d + p->total\_distance then**

**237 f │ │ │ ┌── do;**

**238 f │ │ │ │ put list(d,'miles to',p->city\_name);**

**239 f │ │ │ │ q = null;**

**240 f │ │ │ └── end;**

**241 e │ │ │ else**

**242 e │ │ │ q = q->route\_list;**

**243 e │ │ └── end;**

**244 d │ └── end;**

**245 c └── end print\_route;**

Листинг 12-4 Программа NETWORK (продолжение)

**246 b**

**247 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**248 b /\* Эта процедура освобождает все хранения, выделенные \*/**

**249 b /\* программой при обработке сети. \*/**

**250 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**251 b ┌── free\_all:**

**252 c │ procedure;**

**253 c │ declare**

**254 c │ (p, q) pointer;**

**255 d │ ┌── do p = city\_head**

**256 d │ │ repeat(p->city\_list) while(p^=null);**

**257 e │ │ ┌── do q = p->route\_head**

**258 e │ │ │ repeat(q->route\_list) while(q^=null);**

**259 e │ │ │ free q->route\_node;**

**260 e │ │ └── end;**

**261 d │ │ free p->city\_node;**

**262 d │ └── end;**

**263 c └── end free\_all;**

**264 b**

**265 b end network;**

Листинг 12-4 Программа NETWORK (продолжение)

Ссылки: Справочное руководство разделы 3.4, с 7.1 по 7.8, 8.2

**Конец раздела 12**

# Обработка рекурсии

Рекурсивная обработка происходит, когда активная процедура вызывает себя, или вызывается другой активной процедурой. Существует много задач программирования, которые подходят для такого рода конструкции. Этот раздел содержит три такие задачи. Первые две иллюстрируют фундаментальные понятия, и последняя использует рекурсию в практической задаче.

В рекурсивной процедуре оператор CALL или функциональная ссылка, содержавшаяся в самой процедуре, повторно вызывает процедуру прежде, чем возвратиться к вызову первого уровня. Поэтому необходимо объявлять все такие процедуры с атрибутом RECURSIVE, чтобы, PL/I правильно сохранял и восстанавливал локальные области данных на каждом уровне рекурсивного вызова.

**Примечание**: Для обеспечения совместимости с полным PL/I, вы не должны использовать формальные параметры слева от оператора присваивания в рекурсивной процедуре PL/I.

PL/I не допускает блоки BEGIN в рекурсивных процедурах. Однако он позволяет вложенные процедуры и DO-группы. Приведенные ниже примеры иллюстрируют правильную формулировку рекурсивных процедур.

## Функция факториала

Классическим примером рекурсии служит вычисление функции факториала. Эта функция, широко используемая в математике, является хорошей иллюстрацией, потому что вы можете определить ее итерацией и рекурсией.

Итеративное определение функции факториала следующее

**n! = (n)(n-l)(n-2) ... (2)(1)**

где n! - функция факториала, и n - неотрицательное целое число. Поэтому:

**(n-1)! = (n-1)(n-2) ... (2)(1)**

Вы можете определить функцию факториала, используя рекурсивное отношение:

**n! = n(n-l)!** (по определению **0!** **=** **1**)

Вычисления функции факториала, используя итерацию или рекурсию дают следующие значения:

0! = 1  
 1! = (1) = 1  
 2! = (2)(1) = 2  
 3! = (3)(2)(1) = 6  
 4! = (4)(3)(2)(1) = 24  
 5! = (5)(4)(3)(2)(1) = 120  
 6! = (6)(5)(4)(3)(2)(1) = 720  
 7! = (7)(6)(5)(4)(3)(2)(1) = 5040  
 8! = (8)(7)(6)(5)(4)(3)(2)(1) = 40320  
 9! = (9)(8)(7)(6)(5)(4)(3)(2)(1) = 362880  
10! = (10)(9)(8)(7)(6)(5)(4)(3)(2)(1) = 3628800

*Листинг 13-1* показывает программу под названием IFACT, которая вычисляет значения функции факториала, используя итерации. Переменная F объявлена как элемент данных FIXED BINARY, который накапливает значение факториала максимум до 32767.

*Листинг 13-2* показывает вывод IFACT. IFACT дает правильное значение для функции факториала до 71 - 5040. При этом переменная F переполняется и приводит к некорректным результатам, но вывод продолжается.

**Примечание**: PL/I не устанавливает состояние FIXEDOVERFLOW для двоичных вычислений.

*Листинг 13-3* показывает программу RFACT, которая выполняет аналогичные вычисления функции факториала, используя рекурсию. Для сравнения RFACT использует форму DO-группы REPEAT для проверки условия. RFACT объявляет факториал как рекурсивную процедуру и вызывает процедуру на верхнем уровне в операторе PUT в строке 10. Строка 19 содержит встроенный рекурсивный вызов в операторе RETURN. Факториал возвращается, когда входное значение - ноль. Все другие случаи требуют, чтобы одно или более рекурсивных вычислений факториала привели к результату. Например, 3! производит последовательность вычислений

**factorial(3) = 3\*factorial(2) =  
 factorial(2) 2\*factorial(1)  
 factorial(1) = 1\*factorial(0)  
 factorial(0) = 1  
 = 1 \* 1  
 = 2 \* 1 \* 1  
 = 3 \* 2 \* 1 \* 1**

приводящих в результат к 6. *Листинг 13-4* показывает вывод рекурсивного вычисления факториала, произведенного RFACT. Значения снова переполняются после 5040 из-за точности вычислений.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет функцию факториала (n!), \*/**

**3 a /\* используя итерации. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── ifact:**

**6 b │ procedure options(main);**

**7 b │ declare**

**8 b │ (i, n, F) fixed;**

**9 b │**

**10 c │ ┌── do i = 0 by 1;**

**11 c │ │ F = 1;**

**12 d │ │ ┌── do n = i to 1 by -1;**

**13 d │ │ │ F = n \* F;**

**14 d │ │ └── end;**

**15 c │ │ put edit('factorial(',i,')=',F)**

**16 c │ │ (skip, a,f(2), a, f(7));**

**17 c │ └── end;**

**18 b └── end ifact;**

Листинг 13‑1 Программа IFACT

A>**ifact**

factorial( 0 )= 1

factorial( 1 )= 1

factorial( 2 )= 2

factorial( 3 )= 6

factorial( 4 )= 24

factorial( 5 )= 120

factorial( 6 )= 720

factorial( 7 )= 5040

factorial( 8 )= -25216 ┬ с этого момента

factorial( 9 )= -30336 **│** значения неправильные

factorial( 10 )= 24320 **│**

factorial( 11 )= 5376 **│**

factorial( 12 )= -1024 **│**

factorial( 13 )= -13312 **│**

factorial( 14 )= 10240 **│**

factorial( 15 )= 22528 **│**

factorial( 16 )= -32768 **│**

factorial( 17 )= -32768 **│**

factorial( 18 )= 0 **│**

factorial( 19 )= 0 V

...

Листинг 13‑2 Вывод программы IFACT

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет функцию факториала (n!), \*/**

**3 a /\* используя рекурсию. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── rfact:**

**6 b │ procedure options(main);**

**7 b │ declare**

**8 b │ i fixed;**

**9 c │ ┌── do i = 0 repeat(i+1);**

**10 c │ │ put skip list('factorial(',i,')=',factorial(i));**

**11 c │ └── end;**

**12 b │ stop;**

**13 b │**

**14 b │ ┌── factorial:**

**15 c │ │ procedure(i) returns(fixed) recursive;**

**16 c │ │ declare**

**17 c │ │ i fixed;**

**18 c │ │ if i = 0 then return (1);**

**19 c │ │ return (i \* factorial(i-1));**

**20 c │ └── end factorial;**

**21 b │**

**22 b └── end rfact;**

Листинг 13‑3 Программа RFACT

A>**rfact**

factorial( 0 )= 1

factorial( 1 )= 1

factorial( 2 )= 2

factorial( 3 )= 6

factorial( 4 )= 24

factorial( 5 )= 120

factorial( 6 )= 720

factorial( 7 )= 5040

factorial( 8 )= -25216 ┬ с этого момента

factorial( 9 )= -30336 **│** значения неправильные

factorial( 10 )= 24320 **│**

factorial( 11 )= 5376 **│**

factorial( 12 )= -1024 **│**

factorial( 13 )= -13312 **│**

factorial( 14 )= 10240 **│**

factorial( 15 )= 22528 **│**

factorial( 16 )= -32768 **│**

factorial( 17 )= -32768 **│**

factorial( 18 )= 0 **│**

factorial( 19 )= 0 V

...

Листинг 13‑4 Вывод программы RFACT

## Вычисления FIXED DECINAL и FLOAT BINARY

Приведенные программы вычисления факториала иллюстрируют важный момент об арифметических вычислениях с использованием различных типов данных. *Листинг 13-5* показывает программу под названием DFACT. Это то же рекурсивное вычисление функции факториала, приведенное в RFACT, но использующее данные FIXED DECIMAL с максимальной допустимой точностью. *Листинг 13-6* показывает вывод DFACT. Самое большое значение, вычисленное программой:

**Factorial(17) = 355,687,428,096,000**

При этом система времени выполнения устанавливает состояние FIXEDOVERFLOW, чтобы указать, что десятичное вычисление переполнило значение из максимально возможных 15 цифр.

*Листинг 13-7* показывает программу FFACT, которая вычисляет функцию факториала, используя данные FLOAT BINARY. *Листинг 13-8* показывает вывод FFACT. FFACT может вычислить значение функции вне 17. PL/I усекает количество значительных цифр справа приблизительно до 7 эквивалентным десятичным цифрам. Снова, FFACT заканчивается, когда система времени выполнения устанавливает состояние OVERFLOW, потому что программа производит значение экспоненты, которое не может быть сохранено в представлении с плавающей точкой.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет функцию факториала (n!), \*/**

**3 a /\* используя рекурсию и данные FIXED DECIMAL. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── dfact:**

**6 b │ procedure options(main);**

**7 b │ declare**

**8 b │ i fixed;**

**9 c │ ┌── do i = 0 repeat(i+1);**

**10 c │ │ put skip list('Factorial(',i,')=',factorial(i));**

**11 c │ └── end;**

**12 b │ stop;**

**13 b │**

**14 b │ ┌── factorial:**

**15 c │ │ procedure(i) returns(fixed decimal(15,0))**

**16 c │ │ recursive;**

**17 c │ │ declare**

**18 c │ │ i fixed;**

**19 c │ │**

**20 c │ │ if i = 0 then return (1);**

**21 c │ │ return (decimal(i,15) \* factorial(i-1));**

**22 c │ └── end factorial;**

**23 b │**

**24 b └── end dfact;**

Листинг 13‑5 Программа DFACT

A>**dfact**

Factorial( 0 )= 1

Factorial( 1 )= 1

Factorial( 2 )= 2

Factorial( 3 )= 6

Factorial( 4 )= 24

Factorial( 5 )= 120

Factorial( 6 )= 720

Factorial( 7 )= 5040

Factorial( 8 )= 40320

Factorial( 9 )= 362880

Factorial( 10 )= 3628800

Factorial( 11 )= 39916800

Factorial( 12 )= 479001600

Factorial( 13 )= 6227020800

Factorial( 14 )= 87178291200

Factorial( 15 )= 1307674368000

Factorial( 16 )= 20922789888000

Factorial( 17 )= 355687428096000

Factorial( 18 )=

FIXED OVERFLOW (1)

Traceback: 0007 019F 0018 0000 # 2809 6874 0355 0141

A>

Листинг 13‑6 Вывод программы DFACT

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет функцию факториала (n!), \*/**

**3 a /\* используя рекурсию и данные FLOAT BINARY. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── ffact:**

**6 b │ procedure options(main);**

**7 b │ declare**

**8 b │ i fixed;**

**9 c │ ┌── do i = 0 repeat(i+1);**

**10 c │ │ put skip list('Factorial(',i,')=',factorial(i));**

**11 c │ └── end;**

**12 b │ stop;**

**13 b │**

**14 b │ ┌── factorial:**

**15 c │ │ procedure(i) returns(float) recursive;**

**16 c │ │ declare**

**17 c │ │ i fixed;**

**18 c │ │ if i = 0 then return (1);**

**19 c │ │ return (i \* factorial(i-1));**

**20 c │ └── end factorial;**

**21 b │**

**22 b └── end ffact;**

Листинг 13‑7 Программа FFACT

A>**ffact**

Factorial( 0 )= 1.0000000E+00

Factorial( 1 )= 1.0000000E+00

Factorial( 2 )= 2.0000000E+00

Factorial( 3 )= 6.0000000E+00

Factorial( 4 )= 2.4000000E+01

Factorial( 5 )= 1.2000000E+02

Factorial( 6 )= 7.2000000E+02

Factorial( 7 )= 5.0400000E+03

Factorial( 8 )= 4.0320000E+04

Factorial( 9 )= 3.6288000E+05

Factorial( 10 )= 3.6288000E+06

Factorial( 11 )= 3.9916800E+07

Factorial( 12 )= 4.7900160E+08

Factorial( 13 )= 6.2270210E+09

Factorial( 14 )= 8.7178290E+10

Factorial( 15 )= 1.3076743E+12

Factorial( 16 )= 2.0922780E+13

Factorial( 17 )= 3.5568740E+14

Factorial( 18 )= 6.4023730E+15

Factorial( 19 )= 1.2164510E+17

Factorial( 20 )= 2.4329020E+18

Листинг 13‑8 Вывод программы FFACT

Factorial( 21 )= 5.1090940E+19

Factorial( 22 )= 1.1240007E+21

Factorial( 23 )= 2.5852010E+22

Factorial( 24 )= 6.2044840E+23

Factorial( 25 )= 1.5511211E+25

Factorial( 26 )= 4.0329150E+26

Factorial( 27 )= 1.0888870E+28

Factorial( 28 )= 3.0488830E+29

Factorial( 29 )= 8.8417630E+30

Factorial( 30 )= 2.6525280E+32

Factorial( 31 )= 8.2228390E+33

Factorial( 32 )= 2.6313080E+35

Factorial( 33 )= 8.6833180E+36

Factorial( 34 )=

OVERFLOW (1)

Traceback: 0044 15FE 019B 0000 8608 0B15 FB51 0141

A>

**Листинг 13-8 Вывод программы FFACT (продолжение)**

## Функция Аккермана

Система времени выполнения PL/I поддерживает 512-байтовую область стека для хранения адресов возврата подпрограмм и некоторых временных результатов. При нормальных обстоятельствах эта область стека достаточно большая для обработки нерекурсивной и самой простой рекурсивной процедуры. Однако, программа в этом разделе, иллюстрирует многократную рекурсию, используя глубину стека, которая может превысить 512-байтовое значение по умолчанию.

Функция Аккермана, обозначенная A(m,n) происходит из теории чисел и имеет следующее рекурсивное определение:

**A(m,n) = │ n+1 если m=0, иначе  
 │ A(m-1,1) если n=0, иначе  
 │ A((m-1),A(m,n-1))**

*Листинг 13-9* показывает программу ACK, которая читает два значения для максимума m и n в строке 11, и затем вычисляет функцию для этих значений. *Листинг 13-10* показывает взаимодействие программы. Несмотря на то, что функция Аккермана возвращает значение FIXED BINARY, программа использует встроенную функцию DECIMAL для изменения размера преобразованного поля в операторах PUT в строках 12, 15 и 17.

В этом примере ACK использует опцию STACK в строке 7, чтобы увеличить размер стека времени выполнения от его значения по умолчанию 512 байтов до 2000 байтов.

**Примечание**: опция STACK допустима только с опцией MAIN. Вы должны определить значение опции STACK опытным путем, потому что компилятор не может вычислить глубину рекурсии. Если выделенный размер стека слишком маленький и стек переполняется во время рекурсии, система времени выполнения выводит сообщение

**FREE SPACE OVERWRITE**

и завершит программу.

Обработка множественной рекурсии подобного типа интенсивно использует процессор. Вы должны экспериментировать с несколькими максимальными значениями, и посмотреть, можете ли вы определить, сколько задействуется стека.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет функцию Аккермана A(m,n) и \*/**

**3 a /\* увеличивает размер стека из-за большого количества \*/**

**4 a /\* рекурсивных вызовов. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── ack:**

**7 b │ procedure options(main,stack(2000));**

**8 b │ declare**

**9 b │ (m,maxm,n,maxn) fixed;**

**10 b │ put skip list('Type max m,n: ');**

**11 b │ get list(maxm,maxn);**

**12 b │ put skip**

**13 b │ list(' ',(decimal(n,4) do n=0 to maxn));**

**14 c │ ┌── do m = 0 to maxm;**

**15 c │ │ put skip list(decimal(m,4),':');**

**16 d │ │ ┌── do n = 0 to maxn;**

**17 d │ │ │ put list(decimal(ackermann(m,n),4));**

**18 d │ │ └── end;**

**19 c │ └── end;**

**20 b │ stop;**

**21 b │**

**22 b │ ┌── ackermann:**

**23 c │ │ procedure(m,n) returns(fixed) recursive;**

**24 c │ │ declare (m,n) fixed;**

**25 c │ │ if m = 0 then**

**26 c │ │ return(n+1);**

**27 c │ │ if n = 0 then**

**28 c │ │ return(ackermann(m-1,1));**

**29 c │ │ return(ackermann(m-1,ackermann(m,n-1)));**

**30 c │ └── end ackermann;**

**31 b │**

**32 b └── end ack;**

Листинг 13‑9 Программа ACK

A>**ack**

Type max m,n: **3,5**

0 1 2 3 4 5

0 : 1 2 3 4 5 6

1 : 2 3 4 5 6 7

2 : 3 5 7 9 11 13

3 : 5 13 29 61 125 253

A>

Листинг 13‑10 взаимодействие с программой ACK

## Механизм вычисления арифметических выражений

Одним из практических применений рекурсии является трансляция операторов языка программирования высокого уровня. Это вызвано тем, что большинство языков определено рекурсивно. В блочно-структурированных языках, как PL/I, например, DO-группы и блоки BEGIN и PROCEDURE могут быть вложены, и получающаяся структура легко поддается рекурсивной обработке.

Следующий пример иллюстрирует, как вы можете использовать рекурсию для вычисления арифметических выражений. Вот простое, рекурсивное определение арифметического выражения: выражением является простое число или пара выражений, разделенных знаками +, -, \* или / и заключенные в круглые скобки.

Используя это определение, число 3.6 является выражением, потому что оно - простое число. Очевидно, что

**(3.6 + 6.4)**

выражение, потому что оно состоит из пары выражений, которые являются простыми числами, разделенными знаком +, и заключены в круглые скобки. Кроме того,

**(1.2 \* (3.6 + 6.4))**

допустимое выражение, потому что оно содержит два допустимых выражения 1.2 и (3.6 + 6.4), разделенных знаком \* и заключены в круглые скобки Используя определение, данное выше, последовательности,

**3.6 + 6.4**

**(1.2 + 3.6 + 6.4)**

не являются допустимыми выражениями, потому что первое не заключено в круглые скобки, а второе не является парой выражений в круглых скобках.

Предыдущее определение выражения несколько ограничено. После определения, легко расширить его, включив более сложные выражения.

*Листинг 13-11* показывает программу вычисления выражения под названием EXPR1. Основная обработка происходит между строками 27 и 31, где EXPR1 читает выражение с консоли и выводит вычисленный результат вам назад. *Листинг 13-12* показывает консольное взаимодействие с EXPR1, где пользователь вводит несколько правильно и неправильно сформированных выражений.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет арифметическое выражение, \*/**

**3 a /\* используя рекурсию. Она содержит две процедуры. GNT \*/**

**4 a /\* получает входное выражение, состоящее из отдельных \*/**

**5 a /\* лексем и EXP, которая выполняет рекурсивное \*/**

**6 a /\* вычисление лексем из входной строки. \*/**

**7 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**8 a ┌── expression:**

**9 b │ procedure options(main);**

**10 b │ declare**

**11 b │ sysin file,**

**12 b │ value float,**

**13 b │ token character(10) varying;**

**14 b │**

**15 b │ on endfile(sysin)**

**16 b │ stop;**

**17 b │**

**18 b │ on error(1) /\* преобразование или сигнал \*/**

**19 c │ ┌── begin;**

**20 c │ │ put skip list('Invalid Input at ',token);**

**21 c │ │ get skip;**

**22 c │ │ goto restart;**

**23 c │ └── end;**

Листинг 13‑11 Программа EXPRESSION, использующая анализатор EXPR1

**24 b │**

**25 b │ restart:**

**26 b │**

**27 c │ ┌── do while('1'b);**

**28 c │ │ put skip(3) list('Type expression: ');**

**29 c │ │ value = exp();**

**30 c │ │ put skip list('Value is:',value);**

**31 c │ └── end;**

**32 b │**

**33 b │ ┌── gnt:**

**34 c │ │ procedure;**

**35 c │ │ get list(token);**

**36 c │ └── end gnt;**

**37 b │**

**38 b │ ┌── exp:**

**39 c │ │ procedure returns(float binary) recursive;**

**40 c │ │ declare x float binary;**

**41 c │ │ call gnt();**

**42 c │ │ ┌── if token = '(' then**

**43 d │ │ │ ┌── do;**

**44 d │ │ │ │ x = exp();**

**45 d │ │ │ │ call gnt();**

**46 d │ │ │ │ ┌── if token = '+' then**

**47 d │ │ │ │ │ x = x + exp();**

**48 d │ │ │ │ └── else**

**49 d │ │ │ │ ┌── if token = '-' then**

**50 d │ │ │ │ │ x = x - exp();**

**51 d │ │ │ │ └── else**

**52 d │ │ │ │ ┌── if token = '\*' then**

**53 d │ │ │ │ │ x = x \* exp();**

**54 d │ │ │ │ └── else**

**55 d │ │ │ │ ┌── if token = '/' then**

**56 d │ │ │ │ │ x = x / exp();**

**57 d │ │ │ │ └── else**

**58 d │ │ │ │ signal error(1);**

**59 d │ │ │ │ call gnt();**

**60 d │ │ │ │ if token ^= ')' then**

**61 d │ │ │ │ signal error(1);**

**62 d │ │ │ └── end;**

**63 c │ │ └── else**

**64 c │ │ x = token;**

**65 c │ │ return(x);**

**66 c │ └── end exp;**

**67 b │**

**68 b └── end expression;**

Листинг 13-11 Программа EXPRESSION (продолжение)

### Процедура exp

Основой анализатора выражения является рекурсивная процедура exp. Эта процедура, реализует рекурсивное определение, описанное выше, и разбивает входную часть выражений на части. Процедура GNT[[6]](#footnote-6), получить следующую лексему, читает во входной строке следующий элемент (или лексему), левую или правую круглую скобку, число или один из арифметических операторов. GNT использует оператор GET LIST, таким образом, вы должны разделить каждую лексему символом конца строки или пробелом.

В строке 41 exp вызывает GNT. GNT помещает следующий входную лексему в переменную CHARACTER(10), названную token. Если первый элемент является числом, то ряд проверок в exp передает управление строке 64. Присвоение x автоматически преобразует значение лексемы в значение с плавающей точкой. Затем exp в строке 29 возвращает это преобразованное значение, где EXPR1 сохраняет его в value, и впоследствии записывает его как результат выражения.

Если выражение нетривиально, то exp сканирует ведущую левую круглую скобку в строке 42 и входит в DO-группу в строке 43. EXPR1 сразу вычисляет первое подвыражение, независимо от сложности, и сохранит его в переменную x в строке 44. Затем EXPR1 проверяет token на возникновение +, -, \*, или /. Предположим, например: что token содержит оператор \*. Оператор в строке 53 рекурсивно вызывает процедуру exp для вычисления правой стороны выражения. По возврату результат умножается на значение левой стороны, которое было вычислено ранее. Затем EXPR1 затем проверяет баланс правой скобки, начиная со строки 60, и возвращает полученный продукт как значение exp в строке 64.

### Обработка состояния

EXPR1 выполняет обработку состояний в трех местах. Первый ON-блок в строке 15, прерывает состояние конец файла Ctrl-Z, во входном файле и выполняет оператор STOP. Второй ON-блок в строке 18, получает управление, если происходит ошибка во время преобразования из символьного в представление с плавающей точкой. в присвоении в строке 64. ON-блок отображает token при ошибке, и затем выполняют оператор GET SKIP, чтобы очистить данные до конца строки. Затем управление передается метке restart, где запрашивается другое входное выражение.

При встрече с недопустимым оператором или неуравновешенным выражением, EXPR1 устанавливает состояние. Если оператор не +, -, \*, или /, то EXPR1 вызывает ON-блок в строке 58 сигнализирующий об ошибке. ON-блок снова отображает ошибку и передает управление метке restart. Аналогично, при отсутствии правой скобки в строке 60 устанавливается состояние ERROR(1), чтобы ON-блок сообщил об ошибке и перезапустил программу. При перезапуске программы PL/I отбрасывает информацию о текущем уровне рекурсии.

A>**expr1**

Type expression: **( 4 + 5.2 )**

Value is: 0.920000E+01

Type expression: **4.5e-1**

Value is: 4.499999E-01

Type expression: **( 4 & 5 )**

Invalid input at &

Type expression: **( ( 3 + 4 ) \* ( 10 / 8 ) )**

Value is: 0.875000E+01

Type expression: **( 3 \* 4 )**

Value is: 1.200000E+01

Type Expression: **^Z**

A>

Листинг 13‑12 Взаимодействие с EXPR1

### Внесение улучшений

Анализатор выражений требует пробелы между лексемами во входной строке. Напомним, что [*Раздел 11.2*](#_A_Parse_Function) содержит более усовершенствованную версию GNT.

Мы включили эту расширенную версию GNT в анализатор выражения, и также изменили механизм восстановления после ошибки так, что теперь строка 27 отбрасывает оставшуюся часть текущего ввода при повторном запуске программы. *Листинг 13-13* показывает улучшенную версию под названием EXPR2, и *Листинг 13-14* показывает консольное взаимодействие с этим улучшенным вычислителем выражения.

Даже EXPR2 имеет возможность расширения. Во-первых, можно добавить дополнительные операторы для расширения основных арифметических функций. Также, можно добавить приоритет операторов и исключить требование в отношении явных скобок. Помимо этого, можно добавить имена переменных и операторы присваивания, для превращения программы в интерпретатор BASIC!

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет арифметическое выражение, \*/**

**3 a /\* используя рекурсию. Она содержит расширенную \*/**

**4 a /\* версию процедуры GNT, которая получает выражение, \*/**

**5 a /\* содержащее отдельные лексемы. Затем EXP рекурсивно \*/**

**6 a /\* вычисляет лексемы из входной строки. \*/**

**7 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**8 a**

**9 a ┌── expression:**

**10 b │ procedure options(main);**

**11 b │**

**12 b │ %replace**

**13 b │ true by '1'b;**

**14 b │**

**15 b │ declare**

**16 b │ sysin file,**

**17 b │ value float,**

**18 b │ (token character(10), line character(80)) varying**

**19 b │ static initial('');**

**20 b │**

**21 b │ on endfile(sysin)**

**22 b │ stop;**

**23 b │**

**24 b │ on error(1) /\* преобразование или сигнал \*/**

**25 c │ ┌── begin;**

**26 c │ │ put skip list('Invalid Input at ',token);**

**27 c │ │ token = ''; line = '';**

**28 c │ │ goto restart;**

**29 c │ └── end;**

**30 b │**

**31 b │ restart:**

**32 b │**

Листинг 13‑13 Анализатор выражения EXPR2

**33 c │ ┌── do while('1'b);**

**34 c │ │ put skip(3) list('Type expression: ');**

**35 c │ │ value = exp();**

**36 c │ │ put edit('Value is: ',value) (skip,a,f(10,4));**

**37 c │ └── end;**

**38 b │**

**39 b │ ┌── gnt:**

**40 c │ │ procedure;**

**41 c │ │ declare**

**42 c │ │ i fixed;**

**43 c │ │**

**44 c │ │ line = substr(line,length(token)+1);**

**45 d │ │ do while(true);**

**46 d │ │ if line = '' then**

**47 d │ │ get edit(line) (a);**

**48 d │ │ i = verify(line,' ');**

**49 d │ │ if i = 0 then**

**50 d │ │ line = '';**

**51 d │ │ else**

**52 e │ │ ┌── do;**

**53 e │ │ │ line = substr(line,i);**

**54 e │ │ │ i = verify(line,'0123456789.');**

**55 e │ │ │ ┌── if i = 0 then**

**56 e │ │ │ │ token = line;**

**57 e │ │ │ └── else**

**58 e │ │ │ ┌── if i = 1 then**

**59 e │ │ │ │ token = substr(line,1,1);**

**60 e │ │ │ └── else**

**61 e │ │ │ token = substr(line,1,i-1);**

**62 e │ │ │ return;**

**63 e │ │ └── end;**

**64 d │ │ end;**

**65 c │ └── end gnt;**

**66 b │**

**67 b │ ┌── exp:**

**68 c │ │ procedure returns(float binary) recursive;**

**69 c │ │ declare x float binary;**

**70 c │ │ call gnt();**

**71 c │ │ ┌── if token = '(' then**

**72 d │ │ │ ┌── do;**

**73 d │ │ │ │ x = exp();**

**74 d │ │ │ │ call gnt();**

**75 d │ │ │ │ ┌── if token = '+' then**

**76 d │ │ │ │ │ x = x + exp();**

**77 d │ │ │ │ └── else**

**78 d │ │ │ │ ┌── if token = '-' then**

**79 d │ │ │ │ │ x = x - exp();**

**80 d │ │ │ │ └── else**

**81 d │ │ │ │ ┌── if token = '\*' then**

**82 d │ │ │ │ │ x = x \* exp();**

**83 d │ │ │ │ └── else**

**Листинг 13-13 Анализатор выражения EXPR2 (продолжение)**

**84 d │ │ │ │ ┌── if token = '/' then**

**85 d │ │ │ │ │ x = x / exp();**

**86 d │ │ │ │ └── else**

**87 d │ │ │ │ signal error(1);**

**88 d │ │ │ │ call gnt();**

**89 d │ │ │ │ if token ^= ')' then**

**90 d │ │ │ │ signal error(1);**

**91 d │ │ │ └── end;**

**92 c │ │ └── else**

**93 c │ │ x = token;**

**94 c │ │ return(x);**

**95 c │ └── end exp;**

**96 b │**

**97 b └── end expression;**

**Листинг 13-13 Анализатор выражения EXPR2 (продолжение)**

A>**expr2**

Type expression: **(2 \* 14.5)**

Value is: 29.0000

Type expression: **((2\*3)/(4.3-1.5))**

Value is: 2.1429

Type expression: **zot**

Invalid Input at z

Type expression: **((2\*3)-5)**

Value is: 1.0000

Type expression: **(2 n5)**

Invalid Input at n

Type expression: **^Z**

A>

Листинг 13‑14 Взаимодействие с EXPR2

Ссылки: Справочное руководство разделы с 2.8 по 2.9, 3.1, 3.2, 4.2, с 9.1 по 9.4

**Конец раздела 13**

# Раздельная компиляция

Все программы, представленные до сих пор, являются единственными, полными модулями, несмотря на то, что многие из них содержат одну или несколько внутренних процедур. Часто бывает полезно разбить большие программы на отдельные модули, которые будут впоследствии соединены друг с другом и с библиотекой времени выполнения PL/I.

Есть две причины для отдельной компиляции и связывания программ этим способом. Во-первых, большие программы требуют больше времени для компиляции. Более мелкие сегменты могут быть независимо друг от друга разработаны, протестированы и интегрированы, суммарно тратя меньше времени на компиляцию всего проекта. Большая программа может также превысить объем памяти, доступной для таблицы символов.

Во-вторых, некоторые подпрограммы полезны для вашего собственного прикладного программирования. Вы можете создать свою собственную библиотеку подпрограмм и выборочно соединить их с вашими программами. Наличие такой библиотеки общих подпрограмм значительно уменьшает полное время разработки для какой-либо конкретной программы.

В этом разделе представлены основные сведения, необходимые для связи программных сегментов. Также представлен пример программы, которая компилируется в виде двух отдельных модулей и затем связанных вместе.

## Объявления данных и программ

Вы можете указать отдельным модулям совместно использовать области данных, включая внешний атрибут EXTERNAL в объявлении элемента данных. Например, оператор

**declare x(10) fixed binary external;**

определяет переменную с именем x, состоящую из 20 непрерывных байтов для размещения 10 FIXED BINARY, доступную любым другим модулям, который использует то же объявление.

Аналогичным образом, оператор

**declare  
 1 s external,  
 2 y(10) bit(8),  
 2 z character(9) varying;**

определяет структуру с именем s, занимающую область из 20 байтов, которая доступна любым другим модулям, которые используют то же объявление.

Следующий список обобщает основные правила, которые применяются к объявлению внешних данных:

* Элементы внешних (EXTERNAL) данных доступны в любом блоке, в котором вы объявляете их. Атрибут EXTERNAL переопределяет правила области видимости для внутренних данных.
* Инициализируйте элементы внешних данных только в одном модуле. В дальнейшем, другие модули могут ссылаться на элемент.
* Объявляйте все области внешних (EXTERNAL) данных одинаковой длины во всех модулях, в которых они появляются.
* В 8-разрядных реализациях имена элементов внешних данных должны быть уникальными в первых шести символах, потому что формат редактора связей усекает их, начиная с седьмого символа. В 16-разрядных реализациях это ограничение отсутствует.
* Избегайте использования символов ? в именах переменных, в связи с тем, что этот символ используется в качестве префикса для имен в библиотеке подпрограмм времени выполнения.
* Помните, что PL/I автоматически присваивает атрибут STATIC любому внешнему элементу данных.

## Данные ENTRY

Константы ENTRY и переменные ENTRY представляют собой элементы данных, которые идентифицируют имена процедур и описывают значения их параметров. Константы ENTRY соответствуют внешним процедурам или процедурам, определенным в главной процедуре.

Переменные ENTRY принимают значения констант ENTRY, когда программа непосредственно выполняет оператор присваивания или, неявно, при присвоении фактических параметров формальным в вызове подпрограммы.

Вы вызываете процедуру непосредственно через вызов постоянной ENTRY, или косвенно вызывая переменную ENTRY, хранящую значение адрес процедуры. Как и переменные типа метки, переменные ENTRY можно индексировать.

Программа, показанная в *Листинге 14-1*, иллюстрирует данные ENTRY. Переменная ENTRY f объявленная в строке 8 является массивом, содержащим три константы ENTRY. Начиная выполнение в строке 12, программа инициализирует индексируемые элементы значениями констант a, b и c соответственно. Обратите внимание на то, что константа a соответствует внешне скомпилированной процедуре (см. [*Листинг 3-2*](#Л32)).

В строке 16, DO-группа предлагает ввести значения для отправки в каждую функцию, и затем в строке 19 вызывают каждую функцию один раз с помощью вызова

**f(i)(x)**

в котором первая пара круглых скобок определяет индекс, и вторая включает список фактических аргументов.

Объявление констант и переменных ENTRY подобно файловым константам и переменным. PL/I принимает все формальные параметры, объявленные в качестве переменных с типом ENTRY. Во всех остальных случаях это константа ENTRY, если вы не объявляете ее с ключевым словом VARIABLE.

Следующие правила применяются к объявлениям внешних процедур:

* Вы можете получить доступ к элементам данных с атрибутом EXTERNAL в любой процедуре, где они объявлены EXTERNAL.
* В 8-разрядных реализациях имена элементов внешних данных должны быть уникальными в первых шести символах. (см. [*Раздел 14.1*](#_Объявления_данных_и)) В 16-разрядных реализациях это ограничение отсутствует.
* Избегайте использования символов ? в именах процедур.

**Примечание**: Кроме того, вы должны гарантировать, что каждый параметр точно соответствует объявлению процедуры, и что атрибут RETURNS точно соответствует возвращаемой форме для процедур функций.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа иллюстрирует переменные и \*/**

**3 a /\* константы ENTRY. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a** ┌── **call:**

**6 b** │ **procedure options(main);**

**7 b** │ **declare**

**8 b** │ **f(3) entry(float) returns(float) variable,**

**9 b** │ **a entry(float) returns(float), /\* константа ENTRY \*/**

**10 b** │ **i fixed, x float;**

**11 b** │

**12 b** │ **f(1) = a;**

**13 b** │ **f(2) = b;**

**14 b** │ **f(3) = c;**

**15 b** │

**16 c** │┌── **do i = 1 to 3;**

**17 c** ││ **put skip list('Type x ');**

**18 c** ││ **get list(x);**

**19 c** ││ **put list('f(',i,')=',f(i)(x));**

**20 c** ││ **end;**

**21 b** │└── **stop;**

**22 b** │

**23 b** │┌── **b:**

**24 c** ││ **procedure(x) returns(float); /\* внутренняя процедура \*/**

**25 c** ││ **declare x float;**

**26 c** ││ **return (2\*x + 1);**

**27 c** │└── **end b;**

**28 b** │

**29 b** │┌── **c:**

**30 c** ││ **procedure(x) returns(float); /\* внутренняя процедура \*/**

**31 c** ││ **declare x float;**

**32 c** ││ **return(sin(x));**

**33 c** │└── **end c;**

**34 b** │

**35 b** └── **end call;**

Листинг 14‑1 Иллюстрация констант и переменных ENTRY

## Пример раздельной компиляции

Этот раздел представляет пример программы, состоящей из двух модулей, которые компилируются отдельно и затем соединяются. Эти два модуля называются MAININVT и INVERT, и показаны в *Листингах 14-2* и *14-3*, соответственно. Компиляция каждого из этих модулей и затем соединение их вместе создает программу, которая взаимодействует с консолью для получения множества решений системы одновременных уравнений.

Чтобы понять работу программы сначала рассмотрим следующую систему уравнений с тремя неизвестными:

|  |  |
| --- | --- |
| **a - b + c = 2**  **a + b – c = 0**  **2a – b = 0** | **a - b + c = 3.5**  **a + b – c = 1**  **2a – b = -1** |

Значения

|  |  |
| --- | --- |
| **a = 1**  **b = 2** и  **c = 3** | **a = 2.25**  **b = 5.50**  **c = 6.75** |

составляют допустимые решения этой системы уравнений, потому что:

|  |  |
| --- | --- |
| **1 - 2 + 3 = 2**  **1 + 2 - 3 = 0**  **2\*1 – 2 = 0** | **2.25 - 5.50 + 6.75 = 3.50**  **2.25 + 5.50 - 6.75 = 1**  **2\*2.25 - 5.50 = -1** |

Значения 2,0,0 и 3.5,1,-1 называют векторами решения матрицы. Коэффициенты матрицы:

1 -1 1

1 1 -1

2 -1 0

Модуль MAININVT взаимодействует с консолью, чтобы считать коэффициенты и векторы решения для системы уравнений. Модуль INVERT выполняет обращение матрицы, которая решает систему уравнений.

Существенное различие между этими двумя модулями в заголовке процедуры. Процедура MAININVT - основная программа, потому что она определена с помощью опции MAIN. Процедура INVERT является подпрограммой, вызываемой основной программой. В *Листинге 14-2* объявление, начинающееся в строке 15, определяет invert как константу внешней точки входа (EXTERNAL entry), которая затем вызывается в строке 49.

В строке 21 MAININVT объявляет параметры процедуры invert как матрицу чисел с плавающей точкой, обозначенных maxrow и maxcol. Invert определен с помощью двух дополнительных параметров FIXED(6), но не возвращает значение.

Процедура invert, показанная в *Листинге 14-3*, имеет три формальных параметра, названных a, r и c. Они определены в строке 2 и объявлены в строках 7 и 8. INVERT принимает фактические значения maxrow и maxcol, соответствующие максимальным значениям строки и столбца из %INCLUDE файла, что обозначено символом + после номера строки слева в обоих листингах.

После того, как вы скомпилируете оба модуля, соедините их командой:

**A>link invmat=maininvt,invert**

Редактор связей скомбинирует эти два модуля, выберет необходимые подпрограммы из библиотеки программ времени выполнения и создаст командный файл, названный INVMAT.

Листинг 14-4 показывает взаимодействие с программой INVMAT. В этом демонстрационном взаимодействии пользователь сначала вводит единичную матрицу, чтобы протестировать основные операции. Обратная матрица, произведенная для этого входного значения, является также единичной матрицей.

Затем пользователь водит предыдущую систему уравнений, вместе с двумя векторами решения. выходные значения для этой системы показаны под Solutions: и соответствуют ранее показанным значениям. Второй набор решений соответствует второму вводу вектора решения.

Наконец, пользователь тестирует INVMAT с недопустимым входным размером матрицы, и затем заканчивает программу, вводя нулевой размер строки.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа - основной модуль в программе, которая \*/**

**3 a /\* выполняетобращение матрицы. Она вызывает константу \*/**

**4 a /\* entry INVERT, которая фактически выполняет обращение.\*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── maininvt:**

**7 b │ procedure options(main);**

**8 b │ %replace**

**9 b │ true by '1'b,**

**10 b │ false by '0'b;**

**11+b │ %replace**

**12+b │ maxrow by 26,**

**13+b │ maxcol by 40;**

**14 b │**

**15 b │ declare**

**16 b │ mat(maxrow,maxcol) float binary(24),**

**17 b │ (i,j,n,m) fixed(6),**

**18 b │ var character (26) static initial**

**19 b │ ('abcdefghijklmnopqrstuvwxyz'),**

**20 b │ invert entry**

**21 b │ ((maxrow,maxcol) float(24), fixed(6), fixed(6));**

**22 b │**

**23 b │ put list('Solution of Simultaneous Equations');**

**24 c │ ┌── do while(true);**

**25 c │ │ put skip(2) list('Type rows, columns: ');**

**26 c │ │ get list(n);**

**27 c │ │ if n = 0 then**

**28 c │ │ stop;**

**29 c │ │**

**30 c │ │ get list(m);**

**31 c │ │ ┌── if n > maxrow ! m > maxcol then**

**32 c │ │ │ put skip list('Matrix is Too Large');**

**33 c │ │ └── else**

**34 d │ │ ┌── do;**

**35 d │ │ │ put skip list('Type Matrix of Coefficients');**

**36 d │ │ │ put skip;**

**37 e │ │ │ ┌── do i = 1 to n;**

**38 e │ │ │ │ put list('Row',i,':');**

**39 e │ │ │ │ get list((mat(i,j) do j = 1 to n));**

**40 e │ │ │ └── end;**

**41 d │ │ │**

**42 d │ │ │ put skip list('Type Solution Vectors');**

**43 d │ │ │ put skip;**

**44 e │ │ │ ┌── do j = n + 1 to m;**

**45 e │ │ │ │ put list('Variable',substr(var,j-n,1),':');**

**46 e │ │ │ │ get list((mat(i,j) do i = 1 to n));**

**47 e │ │ │ └── end;**

**48 d │ │ │**

Листинг 14‑2 MAININVT - основной модуль программы обращения матрицы

**49 d │ │ │ call invert(mat,n,m);**

**50 d │ │ │ put skip(2) list('Solutions:');**

**51 e │ │ │ ┌── do i = 1 to n;**

**52 e │ │ │ │ put skip list(substr(var,i,1),'=');**

**53 e │ │ │ │ put edit((mat(i,j) do j = 1 to m-n))**

**54 e │ │ │ │ (f(8,2));**

**55 e │ │ │ └── end;**

**56 d │ │ │**

**57 d │ │ │ put skip(2) list('Inverse Matrix is');**

**58 e │ │ │ ┌── do i = 1 to n;**

**59 e │ │ │ │ put skip edit((mat(i,j) do j = m-n+1 to m))**

**60 e │ │ │ │ (x(3),6f(8,2),skip);**

**61 e │ │ │ └── end;**

**62 d │ │ └── end;**

**63 c │ └── end;**

**64 b │**

**65 b └── end maininvt;**

**Листинг 14-2 MAININVT - основной модуль программы (продолжение)**

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Это внешняя процедура называется MAININVT. \*/**

**3 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**4 a ┌── invert:**

**5 b │ procedure (a,r,c);**

**6+b │ %replace**

**7+b │ maxrow by 26,**

**8+b │ maxcol by 40;**

**9 b │ declare**

**10 b │ (d, a(maxrow,maxcol)) float binary(24),**

**11 b │ (i,j,k,l,r,c) fixed binary(6);**

**12 c │ ┌── do i = 1 to r;**

**13 c │ │ d = a(i,1);**

**14 d │ │ ┌── do j = 1 to c - 1;**

**15 d │ │ │ a(i,j) = a(i,j+1)/d;**

**16 d │ │ └── end;**

**17 c │ │ a(i,c) = 1/d;**

**18 d │ │ ┌── do k = 1 to r;**

**19 d │ │ │ if k ^= i then**

**20 e │ │ │ ┌── do;**

**21 e │ │ │ │ d = a(k,1);**

**22 f │ │ │ │ ┌── do l = 1 to c - 1;**

**23 f │ │ │ │ │ a(k,l) = a(k,l+1) - a(i,l) \* d;**

**24 f │ │ │ │ └── end;**

**25 e │ │ │ │ a(k,c) = - a(i,c) \* d;**

**26 e │ │ │ └── end;**

**27 d │ │ └── end;**

**28 c │ └── end;**

**29 b │**

**30 b └── end invert;**

Листинг 14‑3 Подпрограмма обращения матрицы ININVT

A>**invmat**

Solution of Simultaneous Equations

Type rows, columns: **3,3**

Type Matrix of Coefficients

Row 1 :1 0 0

Row 2 :0 1 0

Row 3 :0 0 1

Type Solution Vectors

Solutions:

a =

b =

c =

Inverse Matrix is

1.00 0.00 0.00

0.00 1.00 0.00

0.00 0.00 1.00

Type rows, columns: **3,5**

Type Matrix of Coefficients

Row 1 :**1 -1 1**

Row 2 :**1 1 -1**

Row 3 :**2 -1 0**

Type Solution Vectors

Variable a :**2 0 0**

Variable b :**3.5 1 -1**

Solutions:

a = 1.00 2.25

b = 2.00 5.50

c = 3.00 6.75

Inverse Matrix is

0.50 0.50 0.00

1.00 1.00 -1.00

1.50 0.50 -1.00

Type rows, columns: **41,0**

Matrix is Too Large

Type rows, columns: **0**

A>

Листинг 14‑4 Взаимодействие с подпрограммой INVMAT

Ссылки: Справочное руководство разделы 3.3.2, с 5.1 по 5.4, 8.2

**Конец раздела 14**

# Десятичные вычисления

Этот раздел объясняет, как PL/I обрабатывает десятичные вычисления, хранит десятичные данные и преобразует типы данных. Изучите этот материал внимательно, поскольку он имеет жизненно важное значение для понимания коммерческих вычислений.

## Сравнение десятичных и двоичных операций

Арифметика, с которой мы лучше всего знакомы, использует десятичную систему счисления. Все операции, такие как сложение и умножение, основаны на числе десять, и включают в себя цифры от нуля до девяти. Компьютеры, однако, выполняют арифметические операции, используя двоичные числа или с основанием 2. Компьютеры используют двоичные числа, потому что 1 и 0 могут быть непосредственно обработаны двухпозиционными электронными ключами, присутствующими в арифметических процессорах.

Большинство языков программирования позволяет писать программы, которые обрабатывают константы и элементы данных с основанием 10 в простых и удобочитаемых формах. Поскольку программы обрабатывают десятичные значения, необходимо преобразовать значения в двоичную форму на вводе, и назад, в десятичной форму при выводе. Эти преобразования из одного типа в другой могут привести к ошибкам усечения, которые недопустимы при коммерческих вычислениях. Таким образом, часто требуется десятичная арифметики, чтобы избежать распространения ошибок в процессе вычислений.

В большинстве языков программирования вы не имеете контроля над внутренним форматом, используемым для обработки чисел. В частности, два самых популярных интерпретатора BASIC для микропроцессоров отличаются, прежде всего, внутренними числовыми форматами. Один использует двоичный с плавающей точкой, в то время как другой выполняет вычисления, используя десятичную арифметику.

Компиляторы языка PASCAL обычно используют двоичные форматы с плавающей и фиксированной точкой с точностью, определенной при реализации, в то время как компиляторы FORTRAN всегда используют двоичные форматы с плавающей или фиксированной точкой.

Вместе с тем, COBOL был разработан для использования в коммерческих приложениях, в которых должны поддерживаться точные значения долларов и центов на протяжении вычислений. В связи с этим, COBOL обрабатывает элементы данных с использованием десятичной арифметики.

PL/I дает вам выбор между представлениями, так что вы можете адаптировать данные в соответствии с потребностями конкретного приложения. PL/I использует элементы данных FIXED DECIMAL для выполнения коммерческих функции и элементы FLOAT BINARY для научной обработки, где скорость вычислений является наиболее важным фактором, и ошибки усечения незначительны или вовсе игнорируются.

Следующие две программы иллюстрируют существенное различие между этими двумя типами данных:

Таблица 15‑1 Различие десятичных и двоичных данных

|  |  |
| --- | --- |
| **┌── decimal\_comp:**  **│ procedure options(main);**  **│ declare**  **│ i fixed,**  **│ t fixed decimal(7,2);**  **│ t = 0;**  **│ ┌── do i = 1 to 10000;**  **│ │ t = t + 3.10;**  **│ └── end;**  **│ put edit(t) (f(10,2));**  **└── end decimal\_comp;** | **┌── binary\_comp:**  **│ procedure options(main);**  **│ declare**  **│ i fixed,**  **│ t float binary(24);**  **│ t = 0;**  **│ ┌── do i = 1 to 10000;**  **│ │ t = t + 3.10;**  **│ └── end;**  **│ put edit(t) (f(10,2));**  **└── end binary\_comp;** |

Обе программы суммируют значение 3.10 в общей сложности 10,000 раз. Единственная разница между этими программами состоит в том, что DECIMAL\_COMP вычисляет результат, используя переменную FIXED DECIMAL, в то время как BINARY\_COMP выполняет вычисления, используя FLOAT BINARY.

DECIMAL\_COMP приводит к корректному результату 31000.00, в то время как BINARY\_COMP производит приближение 30997.30. Разница в 2.70 происходит из-за неустранимых ошибок усечения, которые возникают, когда PL/I преобразует некоторые десятичные константы, такие как 3.10, в их двоичные приближения. DECIMAL\_COMP производит точный результат, поскольку при использовании переменных FIXED DECIMAL преобразования не выполняются.

Эти две программы иллюстрируют более общую проблему. Предположим, что в течение определенного дня, Банк Chase Manhattan обрабатывает 10,000 депозитов 3.10$. Используя программу с данными FLOAT BINARY 3.10$ не могут быть представлены в виде конечного двоичного дробного разложения. Поэтому оно аппроксимируется в форму FLOAT BINARY как 3.099999E+00. Каждое сложение накапливает небольшую ошибку в сумму, приводящую к дополнительным 2.70$, неучтенным в конце дня.

Бывают также ситуации, когда десятичная арифметика производит ошибки усечения, которые могут накапливаться в течение вычислений. Например, дробь 1/3 не может быть представлена как конечная десятичная дробь, и таким образом аппроксимируется как

**0.3333333...**

с максимально возможной точностью. Такие ошибки возникают только когда выполняются явные операции деления.

Трудность с представлениями FLOAT BINARY состоит в том, что некоторые десятичные константы, выраженные как конечные дробные разложения в FIXED DECIMAL, не могут быть записаны в виде конечных двоичных дробей. PL/I обязательно усекает их во время преобразования в форму FLOAT BINARY.

Существуют преимущества и недостатки при выборе арифметики FIXED DECIMAL вместо FLOAT BINARY. Одно из преимуществ арифметики FIXED DECIMAL заключается в том, что она гарантирует, что не происходит потери значащих цифр. Все цифры в вычислении считаются значимыми, так что умножение, например, не усекает цифры в наименее значимых позициях. Еще одно преимущество заключается в том, что арифметика FIXED DECIMAL исключает необходимость манипулирования показателем степени, и эти операции достаточно быстры, по сравнению с альтернативными форматами десятичной арифметики.

Недостаток заключается в том, что вы должны следить за диапазоном значений, которые могут принимать арифметические операнды, потому что все цифры считаются значимыми.

## Формат представления десятичных чисел

Десятичные переменные и константы имеют точность и масштабный коэффициент. Точность - количество цифр в переменной или константе, в то время как масштабный коэффициент - количество цифр в дробной части. Для переменных и констант FIXED DECIMAL, точность не может превышать 15, и масштабный коэффициент не может превышать точность.

Вы можете определить точность и масштабный коэффициент переменной в объявлении переменной. Например,

**declare x fixed decimal(10,3);**

объявляет, что переменная x имеет точность 10 и масштабный коэффициент 3. Компилятор автоматически определяет точность и масштабный коэффициент константы, считая количество цифр в константе, и количество цифр после десятичной точки. Например, константа

**-324.76**

имеет точность 5 и масштабный коэффициент 2.

PL/I внутренне, хранит переменные и константы FIXED DECIMAL как упакованные пары двоично-десятичных чисел (BCD), где каждая цифра BCD занимает старшие или младшие четыре бита каждого байта. Старшая значащая цифра BCD определяет знак числа. Ноль обозначает положительное число, и девять обозначает отрицательное число в дополнительной десятичной форме, как описано ниже. Поскольку PL/I всегда хранит числа в 8-разрядных расположениях байта, может присутствовать дополнительная цифра в конце числа, чтобы выровнять его к границе байта. Например, PL/I хранит число 83.62 как

┌──┬──┬──┐

│08│36│20│

└──┴──┴──┘

где каждая цифра представляет собой 4-битную позицию полубайта в 8-разрядном значении. PL/I сохраняет ведущую пару BCD по нижнему адресу памяти.

PL/I хранит отрицательные числа в дополнительном коде (дополнение до 10) , чтобы упростить арифметические операции. Число дополненное до 10 подобно дополненному до двух двоичному представлению, кроме того что значением дополнения каждой цифры x служит 9-x.

Чтобы получить значение числа дополненного до 10, сформируйте дополнение каждой цифры, вычтя цифру из 9 и добавьте 1 к конечному результату. Таким образом дополнение до 10 числа -2 формируется следующим образом:

**(9 - 2) + 1 = 7 + 1 = 8**

PL/I добавляет разряд знака к числу, которое в результате выглядит как однобайтовое значение:

┌──┐

│98│

└──┘

Рассмотрим пример. Предположим, что вы хотите сложить -2 и +3. PL/I представляет эти числа следующим образом:

┌──┐ ┌──┐ ┌──┐

│98│+│03│=1│01│

└──┘ └──┘ └──┘

PL/I игнорирует целые числа, предшествующие цифре знака и приводит к корректному результату 01. В следующем обсуждении мы показываем отрицательные числа с ведущим знаком ‘-' , учитывая, что внутреннее представление находится в форме дополненной до 9. Таким образом, мы пишем число -2, как

┌──┐

│-2│

└──┘

Необходимость явно сохранять десятичный разряд в памяти отсутствует, потому что компилятор знает точность и масштабный коэффициент для каждой переменной и константы. Перед каждой арифметической операцией скомпилированный код выполняет необходимое выравнивание операндов. В последующих примерах мы показываем позицию десятичной точки, чтобы подчеркнуть эффект выравнивания.

Например, число -324.76 выглядит следующим образом

┌──┬──┬──┐

│-3│24│76│

└──┴──┴──┘

Когда PL/I готовит это значение к арифметической обработке, он сначала загружает его в 8-байтовый стековый фрейм, состоящий из 15 цифр BCD со знаком в старшем разряде. В этом случае, -324.76 выглядит следующим образом

┌──┬──┬──┬──┬──┬──┬──┬──┐

│-0│00│00│00│00│03│24│76│

└──┴──┴──┴──┴──┴──┴──┴──┘

В обычной арифметике, перед началом каждой операции необходимо правильно выровнять операнды для этой операции и, после завершения, вы должны решить, где в результате отображается десятичная точка.

В PL/I компилятор самостоятельно выполняет выравнивание и учитывает положение десятичной точки, но вам полезно представлять, как это происходит, благодаря чему, вы можете избежать состояний переполнения или потери значимости. В некоторых случаях может возникнуть необходимость в принудительном изменении точности или масштабного коэффициента во время вычислений, используя встроенные функции DECIMAL или DIVIDE. Примеры программ, рассматриваемые в следующих разделах, содержат примеры этих функций.

## Сложение и вычитание

В PL/I сложение и вычитание функционально эквивалентны. При вычитании PL/I сначала формирует дополнение до 10 вычитаемого и затем выполняет сложение. Учитывая два числа x и y, с точностью и масштабным коэффициентом (p, q) и (r, s), соответственно, операция происходит следующим образом.

Во-первых, PL/I загружает эти два операнда в стек и затем выравнивает их за счет сдвига операнда с меньшим масштабным коэффициентом влево до десятичной позиции. Учитывая, что масштабный коэффициент x больше, чем масштабный коэффициент y, число y смещается на q-s позиций влево с заполнением нулями наименее значимых позиций.

После выравнивания y имеет точность r+(q-s) и масштабный коэффициент q. PL/I устанавливает состояние FIXEDOVERFLOW, если в процессе выравнивания значимые цифры смещаются в позицию знака.

Вот конкретный пример. Предположим, x = 31465.2437 y = 9343.412, так что x имеет точность p = 9 и масштаб q = 4, в то время как y имеет точность r = 7 и масштабный коэффициент s = 3. До выравнивания числа отображаются следующим образом:

│<──────p=9──────>│

│<─q=4─>│

**x = + 0 0 0 0 0 0 3 1 4 6 5 2 4 3 7**

**↑**

**y = + 0 0 0 0 0 0 0 0 9 3 4 3 4 1 2**

**↑**

│<s=3>│

│<────r=7────>│

PL/I выравнивает y с x, смещаясь на q-s = 4-3 = 1 позицию влево, производя

│<──────p=9──────>│

│<─q=4─>│

**x = + 0 0 0 0 0 0 3 1 4 6 5 2 4 3 7**

**↑**

**y = + 0 0 0 0 0 0 0 9 3 4 3 4 1 2 0**

**↑**

│<──q──>│

│<──r+(q-s)=8──>│

Количество цифр в целой части x равно p-q, в то время как целая часть y содержит r-s цифр

**│<─p-q=5─>│**

**3 1 4 6 5**

**9 3 4 3**

**│<r-s=4>│**

таким образом, сумма должна содержать p-q = 5 цифр в целой части:

**3 1 4 6 5**

**+ 9 3 4 3**

**4 0 8 0 8**

**│<─p-q=5─>│**

Существует вероятность, что некоторые значения могут произвести переполнение, требующие один дополнительный разряд в целой части:

**9 9 9 9 9**

**+ 9 9 9 9 9**

**1 9 9 9 9 8**

**│<(p-q)+1=6>│**

Общее количество цифр в сумме x и y - количество цифр в целой части, (p-q) + 1 = 6, плюс количество цифр в части, заданной q, приводящее к точности

**(p-q) + 1 + q = p + 1**

Учитывая два значения x и y, с произвольной точностью и масштабным коэффициентом, вы можете использовать приведенный выше конкретный случай для получения формы точности и масштабного коэффициента результата. Во-первых, масштаб должен быть большим из заданных q и s

**max(q,s)**

и точность результата должна иметь max(q,s) дробных чисел.

Во-вторых, целая часть x содержит p-q цифры, в то время как целая часть y содержит r-s цифры. Результат содержит больший из p-q и r-s цифр, плюс дробные цифры, вместе с одной цифрой переполнения, в общей сложности

**max(p-q,r-s) + max(q,s) + 1**

позиций символов.

Так как точность не может превысить 15 цифр, точность результата должна быть

**min(15,max(p-q,r-s)+max(q,s)+1)**

цифр.

Точность и масштабный коэффициент результата сложения или вычитания, записанная в виде пары (p', q') следующая:

**p' q'**

**│<─────────────────────────────>│<──────>│**

**(min(15,max(p-q,r-s)+max(q,s)+1), max(q,s))**

Используя предыдущий пример:

│<──────p=9──────>│

│<─q=4─>│

**x = + 0 0 0 0 0 0 3 1 4 6 5 2 4 3 7**

**↑**

**y = + 0 0 0 0 0 0 0 9 3 4 3 4 1 2 0**

**↑**

│<──q──>│

│<──r+(q-s)=8──>│

**x + y = + 0 0 0 0 0 0 4 0 8 0 8 6 5 5 7**

**↑**

│<──4──>│

│<────────10───────>│

Точность (10,4) показанная в схеме получена, , используя выражение

p' q'

│<────────────────────────────>│<──────>│

**(min(15,max(9-4,7-3)+max(4,3)+l), max(4,3)**

или

**min(15,max(5,4)+4+1), 4) = (min(15,10),4) = (10,4)**

## Умножение

Оценка точности и масштабного коэффициента для умножения более проста, чем сложения и вычитания, потому что PL/I не должен выравнивать десятичную точку перед умножением. С учетом двух операндов х и у с точностью и масштабным коэффициентом (p, q) и (r, s) соответственно, PL/I умножает два операнда поразрядно для получения результата.

Так же, как и в обычных ручных вычислениях, количество десятичных разрядов в результате равняется сумме масштабных коэффициентов q и s. Общее количество цифр в результате - сумма точностей этих двух операндов. Для соответствия стандарту PL/I подмножества G, PL/I включает одну дополнительную позицию символа в заключительную точность. Точность и масштабный коэффициент результата (p', q') определяется следующим образом:

p' q'

│<───────────>│<──>│

**(min(15,p+r+1),q+s)**

Предположим что x = 924.5 и y = 862.33 со значениями точности и масштабного коэффициента (4,1) и (5,2):

**x = + 0 0 0 0 0 0 0 0 0 0 0 9 2 4 5**

**↑**

**y = + 0 0 0 0 0 0 0 0 0 0 8 6 2 3 3**

**↑**

Произведение чисел x и y показано с результирующими точностью и масштабным коэффициентом:

**x \* y = + 0 0 0 0 0 0 7 9 7 2 2 4 0 8 5**

**↑**

│<─3─>│

│<────────10───────>│

где PL/I вычисляет точность и масштабный коэффициент следующим образом:

**(min(15,4+5+1),1+2) = (min(15,10),3) = (10,3)**

PL/I устанавливает состояние FIXEDOVERFLOW, если произведение содержит больше чем 15 значимых цифр. В предыдущем разделе, где x = 31465.2437 и y = 9343.412, произведение x\*y имеет точность 17, вызывая FIXEDOVERFLOW.

В данном конкретном случае, необходимо применить функцию DECIMAL, чтобы сократить количество значимых цифр в x или y. Расчет осуществляется как

**DECIMAL(x,9,3) \* y**

который загружает в стек два следующих значения, прежде чем выполнить умножение:

**DECIMAL(x,9,3) = + 0 0 0 0 0 0 0 3 1 4 6 5 2 4 3**

**↑**

**y = + 0 0 0 0 0 0 0 0 9 3 4 3 4 1 2**

**↑**

Точность и масштабный коэффициент произведения следующие:

**x \* y = + 2 9 3 9 9 2 7 2 9 0 2 9 1 1 6**

**↑**

│<────6────>│

│<─────────────15───────────>│

PL/I сначала вычисляет точность как p+r+1 = 16, и затем уменьшает ее до 15 цифр - максимальной точности:

**min(15,p+r+1) = min(15,16) = 15**

При выполнении умножения вы должны убедиться, что соответствующие точности операндов не приводят к переполнению. Вы можете явно объявить точность и масштабный коэффициент переменных, задействованных в вычислении, или использовать функцию DECIMAL для уменьшения точности временного результата.

## Деление

Деление является единственной из четырех основных арифметических операций, которая может вызвать ошибки усечения. Поэтому каждая операция деления дает максимальное значение точности, состоящее из 15 десятичных цифр и результирующего масштабного коэффициента, который зависит от масштабных коэффициентов этих двух операндов.

Предполагая, что x и y имеют точность и масштабный коэффициент (p, q) и (r, s) соответственно, и что x должен быть разделен на y, операция деления происходит следующим образом.

Во-первых, PL/I сдвигает х в крайнее левое положение, вводя 15-р, нулевых значений справа, в результате чего делимое в стеке имеет вид:

│<──────p──────>│<───15-p────>│

│<─q─>│

**x x x x x x x x 0 0 0 0 0 0 0**

**↑**

Затем PL/I смещает десятичную точку x вправо на s, чтобы правильно выровнять десятичную точку в результате, производя следующие операнды:

│<──────p──────>│<───15-p────>│

│<q-s>│<───15-p────>│

**x x x x x x x x 0 0 0 0 0 0 0**

**↑**

│<──────r──────>│

│<─s─>│

**0 0 0 0 0 0 0 y y y y y y y y**

**↑**

Затем значимые цифры y тогда непрерывно делят значимые цифры x, пока операция не сформирует 15 десятичных цифр.

В предыдущей схеме количество дробных цифр, произведенных делением, определяется размещением скорректированной десятичной точки в x. Поле после десятичной точки содержит (q-s) плюс (15-p) позиций, приводя к следующей точности и масштабному коэффициенту для результата деления:

**(15,(q-s)+(15-p))** или **(15,15-p+q-s)**

Предположим x = 31465.243 и y = 9343.41, имеющие значения точности и масштабного коэффициента (8,3) и (6,2), соответственно. Затем значение x, после загрузки в стек имеет вид:

│<──────8──────>│

│<─3─>│

**x = + 0 0 0 0 0 0 0 3 1 4 6 5 2 4 3**

**↑**

Затем PL/I сдвигает значение x в крайнее левое положение и загружает значение y, производя значения:

│<──────8──────>│<──15-8=7───>│

│<─3─>│

**x = + 3 1 4 6 5 2 4 3 0 0 0 0 0 0 0**

**↑**

**y = + 0 0 0 0 0 0 0 0 0 9 3 4 3 4 1**

**↑**

│<2>│

│<────6────>│

Воображаемая десятичная точка смещается вправо на две позиции, чтобы правильно выровнять десятичную точку в результате, произведя

│<──────8──────>│<─────7─────>│

│**1**│

**x = + 3 1 4 6 5 2 4 3 0 0 0 0 0 0 0**

**↑**

**y = + 0 0 0 0 0 0 0 0 0 9 3 4 3 4 1**

**↑**

│<────6────>│

Шесть значимых цифр y делят значимые цифры x со следующим результатом:

│<────────────15─────────────>│

│<───1+7=8───>│

**x/y = + 0 0 0 0 0 0 0 3 3 6 7 6 4 0 1**

**↑**

В этом случае точность и масштабный коэффициент результата следующие:

**(15,15-p+q-s) = (15,15-8+3-2) = (15,8)**

Наиболее важным условием в десятичном делении является генерация достаточного количество цифр в дробной части для выполняемого вычисления. Это выполняется двумя способами.

Во-первых, при выравнивании делимого, PL/I дополняет нулями и обеспечивает 15-p дробных цифр. Таким образом, значения делимого с маленькой точностью генерирует больше дробных цифр.

Во-вторых, если q больше s, то PL/I генерирует (q-s) дополнительные дробные цифры как показано выше. С другой стороны, если делимое содержит меньше дробных цифр, чем делитель, то q меньше s и используются (s-q) дробных цифр.

Случай q = s происходит довольно часто. В этой конкретной ситуации количество дробных цифр полностью зависит от точности делителя и приводит к 15-p дробным цифрам.

Вы также можете захотеть усечь или расширить результат с помощью нулей, используя встроенную функцию DIVIDE во время конкретного вычисления (см. *Справочное руководство языка PL/I, Раздел 4.2.5*), функция имеет форму:

**DIVIDE (x,y,p,q)**

где p и q - литеральные константы. Они могут появиться как выражение или подвыражение в арифметическом вычислении, и иметь тот же эффект как оператор:

**DECIMAL (x/y,p,q)**

Как и выше, x делится на y, но точность и значения масштабного коэффициента устанавливаются (p, q). PL/I выполняет вычисление, как описано, и затем сдвигает полученное значение на соответствующее число разрядов для получения желаемой точности и масштабный коэффициента.

Ссылки: Справочное руководство разделы 3.1.2, 4.2

**Конец раздела 15**

# Коммерческие вычисления

Коммерческие приложения PL/I используют десятичные вычисления. Четыре программы в этом разделе иллюстрируют встроенные функции PL/I, форматы EDIT, включая шаблоны и способ разбиение сложной программы на небольшие, логические завершенные процедуры.

## Простая программа кредитования

*Листинг 16-1* показывает программу LOAN1, которая вычисляет график оплаты кредита с использованием трех входных значений, соответствующих основной сумме кредита (P), годовая процентная ставка (i), и ежемесячному платежу (PMT). LOAN1 непрерывно применяет следующий алгоритм, пока оставшаяся сумма не достигнет нуля, и кредит погашается.

Алгоритм следующий

1. Каждый месяц увеличить сумму основного долга P на величину фиксированной процентной ставки.

**P = P + (i \* P)**

1. Каждый месяц, уменьшить оставшуюся сумму основного долга на сумму платежа

**P = (P + (i \* P)) - PMT**

LOAN1 предполагает, что основная сумма кредита не превышает $999,999,999.99. Таким образом объявление в строке 12 определяет P как переменную FIXED DECIMAL с точностью 11 и масштабным коэффициентом 2. Платеж не превышает $9,999.99, таким образом, PMT объявлен как FIXED DECIMAL с точностью 6 и масштабным коэффициентом 2. Наконец, LOAN1 объявляет процентную ставку i как FIXED DECIMAL(4,2), что позволяет формировать числа такого размера 99.99%. Две переменные m и y соответствуют месяцу и году, начиная с первого месяца первого года.

Программа LOAN1 читает начальные значения между строками 17 и 22. В этом примере LOAN1 не выполняет проверку диапазона. Таким образом, она может принимать отрицательные значения, и может обрабатывать значения платежа, которые не могут погасить кредит. Эти проверки должны быть сделаны в среде реального приложения.

На каждой итерации LOAN1 увеличивает месяц, пока он не достигает 12-го месяца, в этой точке встроенная функция MOD, строка 26, на единицу увеличивает год. Затем LOAN1 в строке 32 отображает текущую сумму долга P и в следующей строке добавляет ежемесячный процент.

В строке 33 LOAN1 выполняет вычисление. Переменная i имеет точность и масштабный коэффициент (4,2), в то время как переменная P имеет точность и масштабный коэффициент (11,2). Поэтому умножение i \* P приводит к временному результату с точностью и масштабным коэффициентом (15,4).

Далее, деление на литеральную константу 1200 является обязательным, поскольку процентная ставка выражается в процентах (деление на 100) за один год (деление на 12). Результат деления (i \* P)/1200 имеет точность 15, потому что константа 1200 имеет точность и масштабный коэффициент (4,0). При делении PL/I вычисляет точность и масштабный коэффициент как (15,15-15+4-0). Наконец, LOAN1 использует встроенную функцию ROUND для округления позиции центов до двух знаков после запятой.

В строке 34 программа LOAN1 проверяет не превышает ли сумма платежа оставшуюся сумму долга за прошлый месяц. Если проверка - истина, в строке 35 сумму платежа заменяется на сумму долга. В строке 36 печатается платеж, и наконец, в строке 37 уменьшается сумму долга на сумму платежа, используя оператор присваивания:

**P = P - PMT;**

*Листинг 16-2* показывает вывод LOAN1 используя начальную ссуду 500$, процентную ставку 14% и оплату 22.10$ в месяц.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа производит расписание платежей по \*/**

**3 a /\* кредиту, используя следующий алгоритм: \*/**

**4 a /\* если P = кредит, i = банковский процент и PMT \*/**

**5 a /\* ежемесячный платеж, то P = (P + (i\*P) - PMT. \*/**

**6 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**7 a ┌── loan1:**

**8 b │ procedure options(main);**

**9 b │ declare**

**10 b │ m fixed binary,**

**11 b │ y fixed binary,**

**12 b │ P fixed decimal(11,2),**

**13 b │ PMT fixed decimal(6,2),**

**14 b │ i fixed decimal(4,2);**

**15 b │**

**16 c │ ┌── do while('1'b);**

**17 c │ │ put skip list('Principal ');**

**18 c │ │ get list(P);**

**19 c │ │ put list('Interest ');**

**20 c │ │ get list(i);**

**21 c │ │ put list('Payment ');**

**22 c │ │ get list(PMT);**

**23 c │ │ m = 0;**

**24 c │ │ y = 0;**

**25 d │ │ ┌── do while (P > 0);**

**26 d │ │ │ if mod(m,12) = 0 then**

**27 e │ │ │ ┌── do;**

**28 e │ │ │ │ y = y + 1;**

**29 e │ │ │ │ put skip list('Year',y);**

**30 e │ │ │ └── end;**

**31 d │ │ │ m = m + 1;**

**32 d │ │ │ put skip list(m,P);**

**33 d │ │ │ P = P + round( i \* P / 1200, 2);**

**34 d │ │ │ if P < PMT**

**35 d │ │ │ then PMT = P;**

**36 d │ │ │ put list(PMT);**

**37 d │ │ │ P = P - PMT;**

**38 d │ │ └── end;**

**39 c │ └── end;**

**40 b │**

**41 b └── end loan1;**

Листинг 16‑1 Программа LOAN1

A>**loan1**

Principal **500**

Interest **14**

Payment **22.10**

Year 1

1 500.00 22.10

2 483.73 22.10

3 467.27 22.10

4 450.62 22.10

5 433.78 22.10

6 416.74 22.10

7 399.50 22.10

8 382.06 22.10

9 364.42 22.10

10 346.57 22.10

11 328.51 22.10

12 310.24 22.10

Year 2

13 291.76 22.10

14 273.06 22.10

15 254.15 22.10

16 235.02 22.10

17 215.66 22.10

18 196.08 22.10

19 176.27 22.10

20 156.23 22.10

21 135.95 22.10

22 115.44 22.10

23 94.69 22.10

24 73.69 22.10

Year 3

25 52.45 22.10

26 30.96 22.10

27 9.22 9.33

Principal **^C**

A>

Листинг 16‑2 Вывод программы LOAN1

## Обычная рента

Листинг 16-3 показывает программу ANNUITY. Учитывая процентную ставку (i) и двух из трех значений, ANNUITY вычисляет текущую стоимость (PV), платеж (PMT) или количество платежных периодов (n) для обычной ренты.

Программа ANNUITY содержит один основной цикл между строками 35 и 80, который читает текущую стоимость, платеж, и ежегодная процентная ставку с консоли. На каждой итерации вы вводите два ненулевых значения и одно нулевое значение, после чего ANNUITY вычисляет значение переменной, которую вы вводите как ноль. ANNUITY сохраняет значения в каждом цикле так, что вы можете ввести запятую, если не хотите изменять значение. В этом примере ANNUITY не проверяет, что значения входных параметров находятся в требуемом диапазоне.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет или текущую стоимость (PV),\*/**

**3 a /\* оплату (PMT) или количество платежных периодов \*/**

**4 a /\* в ренте. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a ┌── annuity:**

**7 b │ procedure options(main);**

**8 b │ %replace**

**9 b │ clear by '^z',**

**10 b │ true by '1'b;**

**11 b │ declare**

**12 b │ PMT fixed decimal(7,2),**

**13 b │ PV fixed decimal(9,2),**

**14 b │ IP fixed decimal(6,6),**

**15 b │ x float binary,**

**16 b │ yi float binary,**

**17 b │ i float binary,**

**18 b │ n fixed;**

**19 b │**

**20 b │ declare**

**21 b │ ftc entry(float binary(24))**

**22 b │ returns(character(17) varying);**

**23 b │**

**24 b │ put list (clear,'^i^iO R D I N A R Y A N N U I T Y');**

**25 b │ put skip(2) list**

**26 b │ ('^iEnter Known Values, or 0, on Each Iteration');**

**27 b │**

**28 b │ on error**

**29 c │ ┌── begin;**

**30 c │ │ put skip list('^iInvalid Data, Re-enter');**

**31 c │ │ goto retry;**

**32 c │ └── end;**

**33 b │**

**34 b │ retry:**

**35 c │ ┌── do while (true);**

**36 c │ │ put skip(3) list('^iPresent Value ');**

**37 c │ │ get list(PV);**

**38 c │ │ put list('^iPayment ');**

**39 c │ │ get list(PMT);**

**40 c │ │ put list('^iInterest Rate ');**

**41 c │ │ get list(yi);**

**42 c │ │ i = yi / 1200;**

**43 c │ │ put list('^iPay Periods ');**

**44 c │ │ get list(n);**

Листинг 16‑3 Программа ANNUITY

**45 c │ │**

**46 c │ │ if PV = 0 | PMT = 0 then**

**47 c │ │ x = 1 - 1/(1+i)\*\*n;**

**48 c │ │**

**49 c │ │ /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**50 c │ │ /\* вычисление текущей стоимости \*/**

**51 c │ │ /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**52 c │ │ if PV = 0 then**

**53 d │ │ ┌── do;**

**54 d │ │ │ PV = PMT \* dec(ftc(x/i),15,6);**

**55 d │ │ │ put edit('^iPresent Value is ',PV)**

**56 d │ │ │ (a,p'$$$,$$$,$$$V.99');**

**57 d │ │ └── end;**

**58 c │ │**

**59 c │ │ /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**60 c │ │ /\* вычисление платежа \*/**

**61 c │ │ /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**62 c │ │ if PMT = 0 then**

**63 d │ │ ┌── do;**

**64 d │ │ │ PMT = PV \* dec(ftc(i/x),15,8);**

**65 d │ │ │ put edit('^iPayment is ',PMT)**

**66 d │ │ │ (a,p'$$,$$$,$$$V.99');**

**67 d │ │ └── end;**

**68 c │ │**

**69 c │ │ /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**70 c │ │ /\* вычисление числа периодов \*/**

**71 c │ │ /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**72 c │ │ if n = 0 then**

**73 d │ │ ┌── do;**

**74 d │ │ │ IP = ftc(i);**

**75 d │ │ │ x = char(PV \* IP / PMT);**

**76 d │ │ │ n = ceil ( - log(1-x)/log(1+i) );**

**77 d │ │ │ put edit('^i',n,' Pay Periods')**

**78 d │ │ │ (a,p'ZZZ9',a);**

**79 d │ │ end;**

**80 c │ └── end;**

**81 b │**

**82 b └── end annuity;**

Листинг 16-3 Программа ANNUITY (продолжение)

*Листинг 16-4* показывает взаимодействие с программой ANNUITY, в которой в качестве ввода используются несколько различных значений.

A>**annuity**

O R D I N A R Y A N N U I T Y

Enter Known Values, or 0, on Each Iteration

Present Value **32000**

Payment **0**

Interest Rate **8.75**

Pay Periods **360**

Payment is $251.74

Present Value **,**

Payment **0**

Interest Rate **,**

Pay Periods **240**

Payment is $282.78

Present Value **0**

Payment **,**

Interest Rate **,**

Pay Periods **,**

Present Value is $31,998.87

Present Value **32000**

Payment **,**

Interest Rate **,**

Pay Periods 0

240 Pay Periods

Present Value **^C**

A>

Листинг 16‑4 Взаимодействие с программой ANNUITY

### Смешанные типы данных

Программа ANNUITY использует данные и FLOAT BINARY и FIXED DECIMAL, потому что она должна выполнить смесь десятичных арифметических расчетов и оценок аналитических функций. Переменные, используемые в программе определены между строками 12 и 18 следующим образом:

* PMT содержит значение платежа, объявлена как FIXED DECIMAL(7,2) и может быть столь же большим как $99,999.99.
* PV содержит текущую стоимость, объявлена как FIXED DECIMAL(9,2) и может быть столь же большой как $99,999,999.99.
* Переменный IP содержит процентную ставку в течение одного месяца и объявлена как FIXED DECIMAL с шестью десятичными разрядами.
* Переменная n содержит количество периодов оплаты, объявлена как FIXED BINARY и может составлять от 1 до 32767.
* Переменные x, yi и i - числа FLOAT BINARY, используются во время вычислений для аппроксимации десятичных чисел с 7 знаками после запятой.

ANNUITY вычисляет неизвестное значение, используя уравнения, показанные ниже, вместо повторения. ANNUITY предполагает, что процентная ставка больше нуля.

Во-первых, текущее значение определяется по формуле:

|  |  |  |
| --- | --- | --- |
|  |  | (1) |

Преобразование уравнения (1) дает:

|  |  |  |
| --- | --- | --- |
|  |  | (2) |

Наконец, решение для n следующее:

|  |  |  |
| --- | --- | --- |
|  |  | (3) |

Следующее выражение появляется в обоих уравнениях (1) и (2):

**1 - 1/(l + i) \*\* n**

Поэтому ANNUITY хранит это значение в переменной x, строка 47, и использует его при вычислении PV, и PMT. x является только десятичным приближением значения, заданного этим выражением.

### Вычисление текущей стоимости PV

Если вы вводите нулевое значение для PV, то ANNUITY выполняет DO-группу находящуюся между строками 53 и 57 и вычисляет PV следующим образом:

**PV = PMT \* dec(ftc(x/i),15,6);**

Строка 20 объявляет ftc как внешнюю подпрограмму. Она является частью библиотеки подпрограмм времени выполнения PL/I, таким образом, ANNUITY должна только объявить, что она является константой точки входа, используемая ей.

Деление x/i приводит к временному результату FLOAT BINARY, который ftc затем преобразует из FLOAT в форму CHARACTER. Например, предположим, что x/i производит значение 3.042455E+01. Тогда ftc(x/i) возвращает 30.42455, которое приемлемо для преобразования в десятичное число. Если PL/I не может преобразовать параметр с плавающей точкой в 15-разрядное десятичное число, ftc устанавливает состояние ERROR(1), обозначающее ошибку преобразования.

Наконец, встроенная функция DECIMAL применена к символьной строке, чтобы преобразовать ее в определенную точность и масштабный коэффициент (15,6). После этого, выполняется умножение и последующее присвоение PMT.

Как эти конкретные значение точности и масштабного коэффициента определяются? Чтобы ответить на вопрос, сначала рассмотрим часть той же программы:

**declare**

**PMT fixed decimal(7,2),**

**PV fixed decimal(9,2),**

**Q fixed decimal(u, v)**

**PV = PMT \* Q;**

где вы должны выбрать соответствующие значения констант для u и v.

PV имеет точность и масштабный коэффициент (9,2), и таким образом должна быть из семи цифр в целой части и двух цифр в дробной части. PL/I генерирует полные семь цифр в целой части, если произведение PMT на Q приводит к какому-либо из точности и масштабного коэффициента:

**(9,2) (10,3) (11,4) (12,5) (13,6) (14,7) (15,8)**

Присвоение PV усекает любые дробные цифры после второго знака после запятой. Поскольку PMT имеет точность и масштабный коэффициент (7,2), Вы можете выбрать Q с точностью и масштабным коэффициентом (15,6). Тогда умножение приводит к результату с точностью и масштабным коэффициентом,

**(min(15,7+15+1),2+6) = (15,8)**

согласно правилам изложенным ранее.

Принимая во внимание выражение со значениями точности и масштабного коэффициента, приведенные ниже

**a = b \* c**

(p,q) (r,s) (u,v)

где p, q, r и s являются константами, вы можете установить точность и масштабный коэффициент c:

**u = 15 v = 15 - (p + q - s)**

Таким образом, используя значения, указанные в исходной программе, точность и масштабный коэффициент Q становится

**v = 15 - (9 + 2 - 2) = 6,** или **(u,v) = (15,6)**

### Вычисление платежа PMT

Если вы вводите ненулевые значения текущей стоимости PV и платежа PMT, то ANNUITY выполняет DO-группу, начинающуюся в строке 63, и вычисляет значение PMT следующим образом:

**PMT = PV \* dec (ftc(i/x),15,8);**

По существу вычисление использует ту же самую методику, показанную в предыдущем примере. Вы должны определить точность и масштабный коэффициент второго операнда в умножении. Фактически вы заинтересованы только в значении масштабного коэффициента, поскольку в качестве точности может быть принято 15. Используя предыдущий анализ, вычислите форму:

**a = b \* c**

(7,2) (9,2) (15,v)

и определите значение v:

**v = 15 - (p + q - s) = 15 -(7 + 2 - 2) = 8**

### Вычисление количества платежных периодов n

Когда вы вводите ненулевые значения PV и PMT, но обнуляете количество периодов, ANNUITY выполняет DO-группу, начинающуюся в строке 73, чтобы вычислить n. Сначала присвоение в строке 74 изменяет значение ежемесячной процентной ставки с FLOAT BINARY на FIXED DECIMAL. Затем, присвоение в строке 75:

**x = char(PV \* IP / PMT);**

сначала вычисляет промежуточный десятичный результат PV \* IP / PMT, потом преобразует результат в CHARACTER, и затем в FLOAT BINARY с помощью присвоения x.

Умножение PV \* IP приводит к временному результату с точностью и масштабным коэффициентом:

**PV \* IP**

(9,2) (7,2)

│ │

└───┬───┘

│

(15,4)

Затем временный результат делится на PMT и результат вычисления в другом временном результате имеет следующие точность и масштабный коэффициент:

**PV\*IP \* PMT**

(15,4) (7,2)

│ │

└────┬────┘

│

(15,2)

потому что, согласно правилам для деления:

**(15,15-p+q-s) = (15,15-15+4-2) = (15,2)**

обеспечивая тем самым два знака после запятой в вычислении.

Промежуточное преобразование в форму CHARACTER необходимо, так как в противном случае PL/I сначала преобразовал бы промежуточный результат в FIXED BINARY, и затем в FLOAT BINARY, приводящий к усечению дробной части. Эта последовательность преобразований необходима для поддержания совместимости с полным языком.

При необходимости, можно сгенерировать дополнительные дробные цифры, применив встроенную функцию DECIMAL после умножения:

**x = char( dec( PV\*P, 11,4 ) / PMT);**

и произвести дробь с точностью и масштабным коэффициентом:

**(15,15-11+4-2) = (15,6)**

ANNUITY использует значение x в выражении в строке 76, чтобы вычислить количество периодов оплаты и применяет функцию CEIL к результату так, чтобы любой неполный месяц обрабатывался как целый месяц в анализе срока оплаты.

Наконец, ANNUITY использует формат редактирования шаблона для выходной записи значений PV, PMT и n.

## Формат графика оплаты кредита

Программа LOAN2, показанная в *Листинге 16-5*, по существу аналогична, представленной в [*Разделе 16.1*](#_Простая_программа_кредитования), но она имеет более продуманный анализ и формат отображения. LOAN2 использует алгоритм, подобный описанному в [*Разделе 16.1*](#_Простая_программа_кредитования_1). Основная обработка происходит между строками 101 и 136, в которых программа ежемесячно увеличивает начальную сумму долга на сумму процентной ставки и ежемесячно снижает ее на сумму платежа, пока задолженность не становится равной нулю.

Четыре листинга, следующие за описанием программы, показывают несколько примеров взаимодействия с LOAN2.

*Листинг 16-6* показывает минимальное отображение, соответствующее ссуде $3000 по 14% процентной ставке с оплатой $144.03. Принят уровень инфляции 0% с начальным платежом в 11/80 и сумма долга на конец года, подлежащая оплате в декабре.

На дисплее отображается основная задолженность, проценты в декабре, ежемесячный платеж, сумма, уплаченная в оплату основного долга в декабре, а также сумму процентов, уплаченных в прошлом месяце финансового года.

*Листинг 16-7* показывает другое исполнение, используя те же значения, что и в первый раз, но с использованием уровня отображения 1. Выход также содержит годовые проценты уплаченные по кредиту за каждый финансовый год, которые будут вычитаться из налогооблагаемого дохода.

*Листинге 16-8* использует те же начальные значения, что и предыдущие примеры, но обеспечивает полное отображение ежемесячного долга, процентов, ежемесячного платежа, оплату за основной кредит и выплаты процентов.

*Листинг 16-9* также показывает тот же кредит, и процентную ставку с корректировкой в долларовом выражении из-за инфляции. В этом примере предполагается уровень инфляции 10%, так что все суммы зависит от курса доллара на момент выдачи кредита.

Для целей налогового учета, отображение, показывающее сумму общих процентов, уплаченных в конце каждого года не масштабируются, и, таким образом, не соответствуют сумме процентов, выплачиваемых в течение года. Если предположить, что уровень инфляции 0%, общий платеж по кредиту составляет 3,456.97, взятый из предыдущего вывода.

Но если предположить, что уровень инфляции составил 10%, общая стоимость кредита в долларах сегодня

**2,457.00**

**+ 374.25**

**2,831.25**

приводит к чистой прибыли 68.75 в течение двух лет!

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет расписание платежей по кредиту, \*/**

**3 a /\* используя тщательно продуманный анализ и формат вывода. \*/**

**4 a /\* Она содержит пять внутренних процедур: DISPLAY, SUMMARY,\*/**

**5 a /\* CURRENT\_YEAR, HEADER и LINE. \*/**

**6 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**7 a loan2:**

**8 b procedure options(main);**

**9 b %replace**

**10 b true by '1'b,**

**11 b false by '0'b,**

**12 b clear by '^z';**

**13 b**

**14 b declare**

**15 b end bit(1),**

**16 b m fixed binary,**

**17 b sm fixed binary,**

**18 b y fixed binary,**

**19 b sy fixed binary,**

**20 b fm fixed binary,**

**21 b dl fixed binary,**

**22 b P fixed decimal(10,2),**

**23 b PV fixed decimal(10,2),**

**24 b PP fixed decimal(10,2),**

**25 b PL fixed decimal(10,2),**

**26 b PMT fixed decimal(10,2),**

**27 b PMV fixed decimal(10,2),**

**28 b INT fixed decimal(10,2),**

**29 b YIN fixed decimal(10,2),**

**30 b IP fixed decimal(10,2),**

**31 b yi fixed decimal(4,2),**

**32 b i fixed decimal(4,2),**

**33 b INF fixed decimal(4,3),**

**34 b ci fixed decimal(15,14),**

**35 b fi fixed decimal(7,5),**

**36 b ir fixed decimal(4,2);**

**37 b**

**38 b declare**

**39 b name character(14) varying static initial('$con'),**

**40 b output file;**

**41 b**

**42 b put list(clear,'^i^iS U M M A R Y O F P A Y M E N T S');**

**43 b**

**44 b on undefinedfile(output)**

**45 c ┌── begin;**

**46 c │ put skip list('^i^icannot write to',name);**

**47 c │ goto open\_output;**

**48 c └── end;**

**49 b**

Листинг 16‑5 Программа LOAN2

**50 b open\_output:**

**51 b put skip(2) list('^i^iOutput File Name ');**

**52 b get list(name);**

**53 b ┌── if name = '$con' then**

**54 b │ open file(output) title('$con') print pagesize(0);**

**55 b └── else**

**56 b open file(output) title(name) print;**

**57 b**

**58 b on error**

**59 c ┌── begin;**

**60 c │ put skip list('^i^iBad Input Data, Retry');**

**61 c │ goto retry;**

**62 c └── end;**

**63 b**

**64 b ┌── retry:**

**65 c │ do while(true);**

**66 c │ put skip(2) list('^i^iPrincipal ');**

**67 c │ get list(PV);**

**68 c │ P = PV;**

**69 c │ put list('^i^iInterest ');**

**70 c │ get list(yi);**

**71 c │ i = yi;**

**72 c │ put list('^i^iPayment ');**

**73 c │ get list(PMV);**

**74 c │ PMT = PMV;**

**75 c │ put list('^i^i%Inflation ');**

**76 c │ get list(ir);**

**77 c │ fi = 1 + ir/1200;**

**78 c │ ci = 1.00;**

**79 c │ put list('^i^iStarting Month ');**

**80 c │ get list(sm);**

**81 c │ put list('^i^iStarting Year ');**

**82 c │ get list(sy);**

**83 c │ put list('^i^iFiscal Month ');**

**84 c │ get list(fm);**

**85 c │ put edit('^i^iDisplay Level ',**

**86 c │ '^i^iYr Results : 0 ',**

**87 c │ '^i^iYr Interest: 1 ',**

**88 c │ '^i^iAll Values : 2 ')**

**89 c │ (skip,a);**

**90 c │ get list(dl);**

**91 c │ if dl < 0 | dl > 2 then**

**92 c │ signal error;**

**93 c │ m = sm;**

**94 c │ y = sy;**

**95 c │ IP = 0;**

**96 c │ PP = 0;**

**97 c │ YIN = 0;**

**98 c │ if name ^= '' then**

**99 c │ put file(output) page;**

**100 c │ call header();**

Листинг 16-5 Программа LOAN2 (продолжение)

**101 d │ ┌── do while (P > 0);**

**102 d │ │ end = false;**

**103 d │ │ INT = round ( i \* P / 1200, 2 );**

**104 d │ │ IP = IP + INT;**

**105 d │ │ PL = P;**

**106 d │ │ P = P + INT;**

**107 d │ │ if P < PMT then**

**108 d │ │ PMT = P;**

**109 d │ │ P = P - PMT;**

**110 d │ │ PP = PP + (PL - P);**

**111 d │ │ INF = ci;**

**112 d │ │ ci = ci / fi;**

**113 d │ │ if P = 0 | dl > 1 | m = fm then**

**114 e │ │ ┌── do;**

**115 e │ │ │ put file(output) skip**

**116 e │ │ │ edit('|',100\*m+y) (a,p'99/99');**

**117 e │ │ │ call display(PL \* INF, INT \* INF,**

**118 e │ │ │ PMT \* INF, PP \* INF, IP \* INF);**

**119 e │ │ └── end;**

**120 d │ │ if m = fm & dl > 0 then**

**121 d │ │ call summary();**

**122 d │ │ m = m + 1;**

**123 d │ │ if m > 12 then**

**124 e │ │ ┌── do;**

**125 e │ │ │ m = 1;**

**126 e │ │ │ y = y + 1;**

**127 e │ │ │ if y > 99 then**

**128 e │ │ │ y = 0;**

**129 e │ │ └── end;**

**130 d │ └── end;**

**131 c │ ┌── if dl = 0 then**

**132 c │ │ call line();**

**133 c │ └── else**

**134 c │ if ^end then**

**135 c │ call summary();**

**136 c └── end retry;**

**137 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**138 b /\* Эта процедура выполняет вывод фактических \*/**

**139 b /\* параметров, переданных ей основной частью \*/**

**140 b /\* программы. \*/**

**141 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**142 b ┌── display:**

**143 c │ procedure(a,b,c,d,e);**

**144 c │ declare**

**145 c │ (a,b,c,d,e) fixed decimal(10,2);**

**146 c │**

**147 c │ put file (output) edit**

**148 c │ ('|',a,'|',b,'|',c,'|',d,'|',e,'|')**

**149 c │ (a,2(2(p'$zz,zzz,zz9v.99',a),**

**150 c │ p'$zzz,zz9.v99',a));**

**151 c └── end display;**

Листинг 16-5 Программа LOAN2 (продолжение)

**152 b**

**153 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**154 b /\* Эта процедура вычисляет суммарные годовые \*/**

**155 b /\* проценты. \*/**

**156 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**157 b ┌── summary:**

**158 c │ procedure;**

**159 c │ end = true;**

**160 c │ call current\_year(IP-YIN);**

**161 c │ YIN = IP;**

**162 c └── end summary;**

**163 b**

**164 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**165 b /\* Эта процедура вычисляет проценты, выплаченные в \*/**

**166 b /\* течение текущего года. \*/**

**167 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**168 b ┌── current\_year:**

**169 c │ procedure(I);**

**170 c │ declare**

**171 c │ yp fixed binary,**

**172 c │ I fixed decimal(10,2);**

**173 c │ yp = y;**

**174 c │ if fm < 12 then**

**175 c │ yp = yp - 1;**

**176 c │ call line();**

**177 c │ put skip file(output) edit**

**178 c │ ('|','Interest Paid During ''',yp,'-''',y,' is ',I,'|')**

**179 c │ (a,x(15),2(a,p'99'),a,p'$$$,$$$,$$9V.99',x(16),a);**

**180 c │ call line();**

**181 c └── end current\_year;**

**182 b**

**183 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**184 b /\* Эта процедура определяет и распечатывает тщательно \*/**

**185 b /\* продуманный формат заголовка. \*/**

**186 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**187 b ┌── header:**

**188 c │ procedure;**

**189 c │ put file(output) list(clear);**

**190 c │ call line();**

**191 c │ put file(output) skip edit**

**192 c │ ('|','L O A N P A Y M E N T S U M M A R Y','|')**

**193 c │ (a,x(19));**

**194 c │ call line();**

**195 c │ put file(output) skip edit**

**196 c │ ('|','Interest Rate',yi,'%','Inflation Rate',ir,'%','|')**

**197 c │ (a,x(15),2(a,p'b99v.99',a,x(6)),x(9),a);**

**198 c │ call line();**

**199 c │ put file(output) skip edit**

**200 c │ ('|Date |',' Principal |','Plus Interest|',' Payment |',**

**201 c │ 'Principal Paid|','Interest Paid |') (a);**

**202 c │ call line();**

**203 c └── end header;**

**204 b**

Листинг 16-5 Программа LOAN2 (продолжение)

**205 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**206 b /\* Эта процедура распечатывает серию пунктирных линий. \*/**

**207 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**208 b ┌── line:**

**209 c │ procedure;**

**210 c │ declare**

**211 c │ i fixed bin;**

**212 c │ put file(output) skip edit**

**213 c │ ('-------','------------',**

**214 c │ ('---------------' do i = 1 to 4)) (a);**

**215 c └── end line;**

**216 b**

**217 b end loan2;**

Листинг 16-5 Программа LOAN2 (продолжение)

### Объявления переменных

Начиная со строки 14, LOAN2 объявляет несколько элементов данных:

* **PV** текущая стоимость, начальный долг;
* **yi** годовая процентная ставка;
* **PMV** ежемесячный платеж;
* **ir** годовой уровень инфляции;
* **sm** стартовый месяц оплаты (1-12);
* **sy** стартовый год оплаты (0-99);
* **fm** отчетный месяц, конец финансового года (1-12);
* **dl** уровень вывода (0-2).

LOAN2 объявляет переменные начального долга и платежа как FIXED DECIMAL(10,2), позволяя значения, столь же большие как $99,999,999.99.

Она также позволяет годовой процентной ставке и годовому уровню инфляции быть столь же большой как 99.99.

Переменные месяца и года, sm, sy и fm являются FIXED BINARY и LOAN2 предполагают, что эти переменные правильно представляют значения месяца и года.

Переменная dl является уровнем вывода и определяет объем информации отображаемый LOAN2 во время конкретной итерации программы. Т.е. 0 производит сокращенный вывод, 1 производит дополнительную информацию, и 2 дает полную трассировку.

LOAN2 также объявляет несколько других переменных, используемых в программе:

|  |  |
| --- | --- |
| * **P** | Сумма долга первоначально устанавливается равной PV, но изменяется во время выполнения. |
| * **PP** | Общая уплаченная сумма. |
| * **PL** | Долг для текущей строки, содержит P с целью отображения. |
| * **PMT** | Платеж изначально устанавливается равным PMV. Изменяется во время выполнения. |
| * **INT** | Начисленный процент за текущий месяц. |
| * **YIN** | Проценты на начало текущего года. |
| * **IP** | Сумма уплаченных процентов. |
| * **i** | Процентная ставка, инициализированная к yi. |
| * **INF** | Процент девальвации курса доллара из-за инфляции. |
| * **ci** | Текущая девальвация из-за инфляции. |
| * **fi** | Коэффициент для вычисления текущей инфляции. |

Для сохранения исходных значений переменных PV и PMV во время вычислений программа использует рабочие переменные суммы долга и оплаты P и PMT. Если вы вводите запятую в ответ на последовавший входной запрос, LOAN2 сохраняет предыдущее значение.

### Выполнение программы

Выполнение программы начинается в строке 42 с символа очистки экрана для CRT ADM-3A Lear Siegler. Этот управляющий символ определен в операторе %REPLACE в строке 12. Если вы используете не ADM-3A, вы можете заменить его корректным символом и перекомпилировать программу.

LOAN2 устанавливает ON-состояние перехвата возможных ошибок в операторе OPEN, строки 54 - 56, и затем запрашивает имя файла выходного отчета. LOAN2 инициализирует имя переменной значением $con, если вы вводите запятую, вместо имени файла или имени устройства, LOAN2 использует консоль в качестве устройства вывода.

Если вы вводите запятую или имя $con в качестве имени выходного файла, то оператор OPEN в строка 54, открывает консоль с нулевым размером страницы. Это означает, что система времени выполнения не выдает символы прогона страницы в конце каждой логической страницы. Иначе, LOAN2 открывает выходной файл или устройство как нормальный файл PRINT, чтобы система времени выполнения помещала символы прогона страницы в выходной файл или отправила их на физическое устройство вывода, обычно принтер, обозначенный $lst.

ON-состояние установленное в строке 58 перехватывает состояния любых возникших ошибок, включая ERROR(1), обозначающее ошибку преобразования данных. Также LOAN2 программно сигнализирует о недопустимости данных в строке 92, если значение dl находится вне диапазона.

LOAN2 не содержит полный набор подпрограмм для проверки ошибок. Чтобы сделать программу коммерчески функциональной, она должна сигнализировать ошибки для всех других недопустимых входных элементов данных, таких как отрицательная процентная ставка. Кроме того, выход за пределы вычислений должен устанавливать состояние FIXED OVERFLOW.

Начиная со строки 67, LOAN2 считывает набор входных значений, а затем инициализирует переменные для каждого набора входных значений, начиная в строке 93. Оператор PUT LIST в строке 99 выполняет прогон страницы, если выходной файл не консоль. Затем в строке 100 печатается заголовок страницы с помощью вызова подпрограммы header. Вы должны сравнить операторы форматирования в подпрограмме header с выходными значениями, показанными в выходных списках.

Основная обработка происходит в DO-группе, начинающей в строке 101, которая неоднократно выполняется, пока сумма долга не уменьшится до нуля. Переменная end указывает, была ли распечатана сводка конца года, строка 159, и таким образом исключает возможность печати дубликата сводки, строка 134.

В строках 103 и 104, LOAN2 вычисляет ежемесячные проценты INT для текущего долга P и суммирует его в IP. LOAN2 сохраняет текущий долг для последующего отображения в PL, а затем добавляет ежемесячные проценты к основной сумме долга. Если платеж превышает оставшуюся основную сумму долга в строке 107, LOAN2 уменьшает платеж, чтобы покрыть этот остаток. Затем он уменьшает основную сумму на суммы платежа, в конечном счете производя нулевое значение, если первоначальный платеж достаточен для выплаты кредита. Затем в строках 111 и 112 LOAN2 суммирует оплаченный долг и вычисляет уровень инфляции.

### Форматы отображения

The decision logic for displaying the current computation is somewhat complicated because LOAN2 has three display formats. If it is the last iteration, the principal P is zero, you select the full display format, dl > 1, or the current month is the end of the fiscal year, m = fm, then LOAN2 writes the current computation between lines 114 and 118.

The Picture format p'99/99' displays the month and year, where 100\*m + y produces a four-digit number to match this format. For example, if m = 11 and y = 64, then,

100 \* m + y = 100 \* 11 + 64 = 1164

1164 appears as 11164, when printed using the given Picture format.

The DISPLAY subroutine actually performs the output function, based upon the six actual parameters listed in the CALL statement on line 117. The main program first adjusts each argument, by the current inflation rate INF, and then passes it to DISPLAY. If the inflation rate is set to 0%, the value of INF is 1.00 at this point in the computation.

The body of the display subroutine, listed between lines 142 and 151 could be included in the line subroutine because there is only one call to display. However, display illustrates FIXED DECIMAL parameter passing mechanisms and serves to break the program into smaller, more readable, segments. Again, you should compare the format specifications in the display subroutine with the actual program output.

The statement on line 120 then checks for the end of fiscal year, m = fm, and, if the display mode is either 1 or 2, LOAN2 prints a yearly interest summary using the summary subroutine. Summary in turn, calls the current\_year subroutine to write the yearly interest paid, IP-YIN. The assignment on line 161 retains the base value for the next year's display in YIN.

If the fiscal year does not end in December, fm< 12, current\_year splits the interest rate payment between two calendar years, yp = y - 1. Again, you could combine current\_year with the summary subroutine without changing the overall program logic.

The end of the main loop, between lines 131 and 136, contains statements that finalize the report. If you select the abbreviated display format, dl = 0, the CALL statement on line 132 invokes LINE and prints a line of dashes to complete the display.

Otherwise, LOAN2 checks to ensure there have been intervening output lines (end). If there have been, it prints an interest summary on line 130. Finally, control returns to the top of the DO-group, and LOAN2 reads additional input parameters.

A>**loan2**

S U M M A R Y O F P A Y M E N T S

Output File Name **,**

Principal **3000**

Interest **14**

Payment **144.03**

%Inflation **0**

Starting Month **11**

Starting Year **80**

Fiscal Month **12**

Display Level

Yr Results : 0

Yr Interest: 1

All Values : 2 **0**

-------------------------------------------------------------------------------

| L O A N P A Y M E N T S U M M A R Y |

-------------------------------------------------------------------------------

| Interest Rate 14.00% Inflation Rate 00.00% |

-------------------------------------------------------------------------------

|Date | Principal |Plus Interest| Payment |Principal Paid|Interest Paid |

-------------------------------------------------------------------------------

|12/80|$ 2,890.97|$ 33.73|$ 144.03|$ 219.33|$ 68.73|

|12/81|$ 1,479.02|$ 17.26|$ 144.03|$ 1,647.75|$ 368.67|

|11/82|$ 0.25|$ 0.00|$ 0.25|$ 3,000.00|$ 456.97|

-------------------------------------------------------------------------------

Листинг 16‑6 Первое взаимодействие с LOAN2

Principal ,

Interest ,

Payment ,

%Inflation ,

Starting Month ,

Starting Year ,

Fiscal Month ,

Display Level

Yr Results : 0

Yr Interest: 1

All Values : 2 1

-------------------------------------------------------------------------------

| L O A N P A Y M E N T S U M M A R Y |

-------------------------------------------------------------------------------

| Interest Rate 14.00% Inflation Rate 00.00% |

-------------------------------------------------------------------------------

|Date | Principal |Plus Interest| Payment |Principal Paid|Interest Paid |

-------------------------------------------------------------------------------

|12/80|$ 2,890.97|$ 33.73|$ 144.03|$ 219.33|$ 68.73|

-------------------------------------------------------------------------------

| Interest Paid During '80-'80 is $68.73 |

-------------------------------------------------------------------------------

|12/81|$ 1,479.02|$ 17.26|$ 144.03|$ 1,647.75|$ 368.67|

-------------------------------------------------------------------------------

| Interest Paid During '81-'81 is $299.94 |

-------------------------------------------------------------------------------

|11/82|$ 0.25|$ 0.00|$ 0.25|$ 3,000.00|$ 456.97|

-------------------------------------------------------------------------------

| Interest Paid During '82-'82 is $88.30 |

-------------------------------------------------------------------------------

Листинг 16‑7 Второе взаимодействие с LOAN2

Principal ,

Interest ,

Payment ,

%Inflation ,

Starting Month ,

Starting Year ,

Fiscal Month ,

Display Level

Yr Results : 0

Yr Interest: 1

All Values : 2 2

-------------------------------------------------------------------------------

| L O A N P A Y M E N T S U M M A R Y |

-------------------------------------------------------------------------------

| Interest Rate 14.00% Inflation Rate 00.00% |

-------------------------------------------------------------------------------

|Date | Principal |Plus Interest| Payment |Principal Paid|Interest Paid |

-------------------------------------------------------------------------------

|11/80|$ 3,000.00|$ 35.00|$ 144.03|$ 109.03|$ 35.00|

|12/80|$ 2,890.97|$ 33.73|$ 144.03|$ 219.33|$ 68.73|

-------------------------------------------------------------------------------

| Interest Paid During '80-'80 is $68.73 |

-------------------------------------------------------------------------------

|01/81|$ 2,780.67|$ 32.44|$ 144.03|$ 330.92|$ 101.17|

|02/81|$ 2,669.08|$ 31.14|$ 144.03|$ 443.81|$ 132.31|

|03/81|$ 2,556.19|$ 29.82|$ 144.03|$ 558.02|$ 162.13|

|04/81|$ 2,441.98|$ 28.49|$ 144.03|$ 673.56|$ 190.62|

|05/81|$ 2,326.44|$ 27.14|$ 144.03|$ 790.45|$ 217.76|

|06/81|$ 2,209.55|$ 25.78|$ 144.03|$ 908.70|$ 243.54|

|07/81|$ 2,091.30|$ 24.40|$ 144.03|$ 1,028.33|$ 267.94|

|08/81|$ 1,971.67|$ 23.00|$ 144.03|$ 1,149.36|$ 290.94|

|09/81|$ 1,850.64|$ 21.59|$ 144.03|$ 1,271.80|$ 312.53|

|10/81|$ 1,728.20|$ 20.16|$ 144.03|$ 1,395.67|$ 332.69|

|11/81|$ 1,604.33|$ 18.72|$ 144.03|$ 1,520.98|$ 351.41|

|12/81|$ 1,479.02|$ 17.26|$ 144.03|$ 1,647.75|$ 368.67|

-------------------------------------------------------------------------------

| Interest Paid During '81-'81 is $299.94 |

-------------------------------------------------------------------------------

|01/82|$ 1,352.25|$ 15.78|$ 144.03|$ 1,776.00|$ 384.45|

|02/82|$ 1,224.00|$ 14.28|$ 144.03|$ 1,905.75|$ 398.73|

|03/82|$ 1,094.25|$ 12.77|$ 144.03|$ 2,037.01|$ 411.50|

|04/82|$ 962.99|$ 11.23|$ 144.03|$ 2,169.81|$ 422.73|

|05/82|$ 830.19|$ 9.69|$ 144.03|$ 2,304.15|$ 432.42|

|06/82|$ 695.85|$ 8.12|$ 144.03|$ 2,440.06|$ 440.54|

|07/82|$ 559.94|$ 6.53|$ 144.03|$ 2,577.56|$ 447.07|

|08/82|$ 422.44|$ 4.93|$ 144.03|$ 2,716.66|$ 452.00|

|09/82|$ 283.34|$ 3.31|$ 144.03|$ 2,857.38|$ 455.31|

|10/82|$ 142.62|$ 1.66|$ 144.03|$ 2,999.75|$ 456.97|

|11/82|$ 0.25|$ 0.00|$ 0.25|$ 3,000.00|$ 456.97|

-------------------------------------------------------------------------------

| Interest Paid During '82-'82 is $88.30 |

-------------------------------------------------------------------------------

Листинг 16‑8 Третье взаимодействие с LOAN2

Principal ,

Interest ,

Payment ,

%Inflation 10

Starting Month ,

Starting Year ,

Fiscal Month 10

Display Level

Yr Results : 0

Yr Interest: 1

All Values : 2 2

-------------------------------------------------------------------------------

| L O A N P A Y M E N T S U M M A R Y |

-------------------------------------------------------------------------------

| Interest Rate 14.00% Inflation Rate 10.00% |

-------------------------------------------------------------------------------

|Date | Principal |Plus Interest| Payment |Principal Paid|Interest Paid |

-------------------------------------------------------------------------------

|11/80|$ 3,000.00|$ 35.00|$ 144.03|$ 109.03|$ 35.00|

|12/80|$ 2,864.95|$ 33.42|$ 142.73|$ 217.35|$ 68.11|

|01/81|$ 2,733.39|$ 31.88|$ 141.58|$ 325.29|$ 99.45|

|02/81|$ 2,602.35|$ 30.36|$ 140.42|$ 432.71|$ 129.00|

|03/81|$ 2,471.83|$ 28.83|$ 139.27|$ 539.60|$ 156.77|

|04/81|$ 2,341.85|$ 27.32|$ 138.12|$ 645.94|$ 182.80|

|05/81|$ 2,212.44|$ 25.81|$ 136.97|$ 751.71|$ 207.08|

|06/81|$ 2,083.60|$ 24.31|$ 135.82|$ 856.90|$ 229.65|

|07/81|$ 1,955.36|$ 22.81|$ 134.66|$ 961.48|$ 250.52|

|08/81|$ 1,829.70|$ 21.34|$ 133.65|$ 1,066.60|$ 269.99|

|09/81|$ 1,702.58|$ 19.86|$ 132.50|$ 1,170.05|$ 287.52|

|10/81|$ 1,576.11|$ 18.38|$ 131.35|$ 1,272.85|$ 303.41|

-------------------------------------------------------------------------------

| Interest Paid During '80-'81 is $332.69 |

-------------------------------------------------------------------------------

|11/81|$ 1,451.91|$ 16.94|$ 130.34|$ 1,376.48|$ 318.02|

|12/81|$ 1,326.68|$ 15.48|$ 129.19|$ 1,478.03|$ 330.69|

|01/82|$ 1,203.50|$ 14.04|$ 128.18|$ 1,580.64|$ 342.16|

|02/82|$ 1,079.56|$ 12.59|$ 127.03|$ 1,680.87|$ 351.67|

|03/82|$ 957.46|$ 11.17|$ 126.02|$ 1,782.38|$ 360.06|

|04/82|$ 835.87|$ 9.74|$ 125.01|$ 1,883.39|$ 366.92|

|05/82|$ 714.79|$ 8.34|$ 124.00|$ 1,983.87|$ 372.31|

|06/82|$ 594.25|$ 6.93|$ 123.00|$ 2,083.81|$ 376.22|

|07/82|$ 474.26|$ 5.53|$ 121.99|$ 2,183.19|$ 378.66|

|08/82|$ 354.84|$ 4.14|$ 120.98|$ 2,281.99|$ 379.68|

|09/82|$ 236.02|$ 2.75|$ 119.97|$ 2,380.19|$ 379.27|

|10/82|$ 117.80|$ 1.37|$ 118.96|$ 2,477.79|$ 377.45|

-------------------------------------------------------------------------------

| Interest Paid During '81-'82 is $124.28 |

-------------------------------------------------------------------------------

|11/82|$ 0.20|$ 0.00|$ 0.20|$ 2,457.00|$ 374.25|

-------------------------------------------------------------------------------

| Interest Paid During '81-'82 is $0.00 |

-------------------------------------------------------------------------------

Листинг 16‑9 Четвертое взаимодействие с LOAN2

## Расчет графиков амортизации

Заключительный пример коммерческой обработки включает расчет графиков амортизации. *Листинг 16-10* показывает программу под названием DEPREC, которая читает несколько входных значений и распечатывает таблицу вывода в соответствии с одним из трех различных графиков амортизации:

* линейный
* суммы лет
* двойного снижения

Программа также учитывает амортизационную премию в течение первого года, уменьшение налогооблагаемого дохода за счет налога с продаж, а также инвестиционного налогового кредита на новое или бывшее в употреблении оборудование.

*Листинги с 16-11* *до 16-15* иллюстрируют примеры взаимодействия с DEPREC, используя различные входные параметры.

### Общие алгоритмы

DEPREC использует следующие общие алгоритмы:

* Инвестиционный налоговый кредит (ITC) принимается равным 10% отпускной цены, применяется к полной стоимости нового оборудования, или до $100,000 в случае ранее используемого оборудования. (См. оператор %REPLACE в строке 11.)
* Амортизационная премия, принимается равной 10% отпускной цены максимум до $2,000. (См. строки 12 и 13.)
* Во всех трех графиках амортизации, сумма амортизации берется как разница между ценой продажи, за вычетом амортизационной премии и остаточной стоимости оборудования.
* Во всех трех графиках, сумма амортизации, вычисленная для первого года, распределяется пропорционально по месяцам на весь оставшийся период финансового года, не включая амортизационную премию.
* В линейной амортизации сумма, сумма амортизации равномерно распределяется по числу лет, в которых происходит амортизация.
* Для суммы лет значения лет суммируются, начиная с 1 до числа лет, в которые происходит амортизация:

**ys= 1 + 2 + 3 +...+ years**

* Амортизации распределяется по общему количеству лет, вычисляя years/ys умноженное на величину амортизации за первый год, (years-1)/ys умноженный на остаток в течение второго года, и т.д., до последнего года, в котором берется 1/ys, умножается на оставшееся значение амортизации.
* Для двойного снижения годовая амортизация рассчитывается, как остаточная стоимость деленная на количество лет, которое затем умножается на 2, для нового оборудования, или 1.5 если оборудование ранее использовалось.

DEPREC сначала читает отпускную цену, остаточную стоимость, процент налога с продаж, процент подоходного налога, количество месяцев, оставшихся в текущем финансовом году и число лет амортизации оборудования. Затем она спрашивает, новое оборудование или подержанное, и читает код графика амортизации для последующего отчета.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа вычисляет три вида графиков \*/**

**3 a /\*** амортизации: straight\_line, sum\_of\_the\_years и **\*/**

**4 a /\* double\_declining. \*/**

**5 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**6 a depreciate:**

**7 b procedure options(main);**

**8 b %replace**

**9 b clear\_screen by '^z',**

**10 b indent by 15,**

**11 b ITC\_rate by .1,**

**12 b bonus\_rate by .1,**

**13 b bonus\_max by 2000;**

**14 b**

**15 b declare**

**16 b selling\_price decimal(8,2),**

**17 b adjusted\_price decimal(8,2),**

**18 b residual\_value decimal(8,2),**

**19 b year\_value decimal(8,2),**

**20 b depreciation\_value decimal(8,2),**

**21 b total\_depreciation decimal(8,2),**

**22 b book\_value decimal(8,2),**

**23 b tax\_rate decimal(3,2),**

**24 b sales\_tax decimal(8,2),**

**25 b tax\_bracket decimal(2),**

**26 b FYD decimal(8,2),**

**27 b ITC decimal(8,2),**

**28 b bonus\_dep decimal(8,2),**

**29 b months\_remaining decimal(2),**

**30 b new character(4),**

**31 b factor decimal(2,1),**

**32 b years decimal(2),**

**33 b year\_sum decimal(3),**

**34 b current\_year decimal(2),**

**35 b select\_sched character(1);**

**36 b**

**37 b declare**

**38 b copy\_to\_list character(4),**

**39 b output file variable,**

**40 b (sysprint, list) file;**

**41 b**

**42 b declare**

**43 b schedules character(3) static initial ('syd'),**

**44 b schedule (0:3) entry variable;**

**45 b**

**46 b schedule (0) = error;**

**47 b schedule (1) = straight\_line;**

**48 b schedule (2) = sum\_of\_years;**

**49 b schedule (3) = double\_declining;**

Листинг 16‑10 Программа DEPREC

**50 b**

**51 b open file (sysprint) stream print pagesize(0)**

**52 b title ('$con');**

**53 b**

**54 c ┌── do while('1'b);**

**55 c │ put list(clear\_screen,'^i^i^iDepreciation Schedule');**

**56 c │ put skip(3) list('^i^iSelling Price? ');**

**57 c │ get list(selling\_price);**

**58 c │ put list('^i^iResidual Value? ');**

**59 c │ get list(residual\_value);**

**60 c │ put list('^i^iSales Tax (%)? ');**

**61 c │ get list(tax\_rate);**

**62 c │ put list('^i^iTax Bracket(%)? ');**

**63 c │ get list(tax\_bracket);**

**64 c │ put list('^i^iProRate Months? ');**

**65 c │ get list(months\_remaining);**

**66 c │ put list('^i^iHow Many Years? ');**

**67 c │ get list(years);**

**68 c │ put list('^i^iNew? (yes/no) ');**

**69 c │ get list(new);**

**70 c │ put edit('^i^iSchedule:',**

**71 c │ '^i^iStraight (s)',**

**72 c │ '^i^iSum-of-Yrs (y)',**

**73 c │ '^i^iDouble Dec (d)? ') (a,skip);**

**74 c │ get list(select\_sched);**

**75 c │ put list('^i^iList? (yes/no) ');**

**76 c │ get list(copy\_to\_list);**

**77 c │ if copy\_to\_list = 'yes' then**

**78 c │ open file(list) stream print title('$lst');**

**79 c │ factor = 1.5;**

**80 c │ if new = 'yes' then**

**81 c │ factor = 2.0;**

**82 c │ sales\_tax = decimal(selling\_price\*tax\_rate,12,2)/100+.005;**

**83 c │ ┌── if new = 'yes' | selling\_price <= 100000.00 then**

**84 c │ │ ITC = selling\_price \* ITC\_rate;**

**85 c │ └── else**

**86 c │ ITC = 100000 \* ITC\_rate;**

**87 c │ bonus\_dep = selling\_price \* bonus\_rate;**

**88 c │ if bonus\_dep > bonus\_max then**

**89 c │ bonus\_dep = bonus\_max;**

**90 c │ put list(clear\_screen);**

**91 c │ call display(sysprint);**

**92 c │ if copy\_to\_list = 'yes' then**

**93 c │ call display(list);**

**94 c │ put skip list('^i^i^i Type RETURN to Continue');**

**95 c │ get skip(2);**

**96 c └── end;**

Листинг 16-10 Программа DEPREC (продолжение)

**97 b**

**98 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**99 b /\* Эта процедура выводит на экран различные графики \*/**

**100 b /\* амортизации. Она вызывает конкретный график \*/**

**101 b /\* индексом массива констант точек входа. \*/**

**102 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**103 b ┌── display:**

**104 c │ procedure(f);**

**105 c │ declare**

**106 c │ f file;**

**107 c │ output = f;**

**108 c │ call schedule (index (schedules,select\_sched));**

**109 c └── end display;**

**110 b**

**111 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**112 b /\* Это глобальная подпрограмма восстановления после ошибки. \*/**

**113 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**114 b ┌── error:**

**115 c │ procedure;**

**116 c │ put file (output) edit('Invalid Schedule - Enter s, y, or d')**

**117 c │ (page,column(indent),x(8),a);**

**118 c │ call line();**

**119 c └── end error;**

**120 b**

**121 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**122 b /\* Эта процедура вычисляет амортизацию straight\_line. \*/**

**123 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**124 b ┌── straight\_line:**

**125 c │ procedure;**

**126 c │ adjusted\_price = selling\_price - bonus\_dep;**

**127 c │ put file (output) edit('S T R A I G H T L I N E')**

**128 c │ (page,column(indent),x(14),a);**

**129 c │ call header();**

**130 c │ depreciation\_value = adjusted\_price - residual\_value;**

**131 c │ book\_value = adjusted\_price;**

**132 c │ total\_depreciation = 0;**

**133 d │ ┌── do current\_year = 1 to years;**

**134 d │ │ year\_value = decimal(depreciation\_value/years,8,2) + .005;**

**135 d │ │ if current\_year = 1 then**

**136 e │ │ ┌── do;**

**137 e │ │ │ year\_value = year\_value \* months\_remaining / 12;**

**138 e │ │ │ FYD = year\_value;**

**139 e │ │ └── end;**

**140 d │ │ depreciation\_value = depreciation\_value - year\_value;**

**141 d │ │ total\_depreciation = total\_depreciation + year\_value;**

**142 d │ │ book\_value = adjusted\_price - total\_depreciation;**

**143 d │ │ call print\_line();**

**144 d │ └── end;**

**145 c │ call summary();**

**146 c └── end straight\_line;**

Листинг 16-10 Программа DEPREC (продолжение)

**147 b**

**148 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**149 b /\* Эта процедура вычисляет амортизацию на основе \*/**

**150 b /\* sum\_of\_the\_years. \*/**

**151 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**152 b ┌── sum\_of\_years:**

**153 c │ procedure;**

**154 c │ adjusted\_price = selling\_price - bonus\_dep;**

**155 c │ put file (output) edit('S U M O F T H E Y E A R S')**

**156 c │ (page,column(indent),x(11),a);**

**157 c │ call header();**

**158 c │ depreciation\_value = adjusted\_price - residual\_value;**

**159 c │ book\_value = adjusted\_price;**

**160 c │ total\_depreciation = 0;**

**161 c │ year\_sum = 0;**

**162 d │ ┌── do current\_year = 1 to years;**

**163 d │ │ year\_sum = year\_sum + current\_year;**

**164 d │ └── end;**

**165 c │**

**166 d │ ┌── do current\_year = 1 to years;**

**167 d │ │ year\_value = decimal(depreciation\_value \***

**168 d │ │ (years - current\_year + 1),12,2)/ year\_sum + .005;**

**169 d │ │ if current\_year = 1 then**

**170 e │ │ ┌── do;**

**171 e │ │ │ year\_value = year\_value \* months\_remaining / 12;**

**172 e │ │ │ FYD = year\_value;**

**173 e │ │ └── end;**

**174 d │ │ depreciation\_value = depreciation\_value - year\_value;**

**175 d │ │ total\_depreciation = total\_depreciation + year\_value;**

**176 d │ │ book\_value = adjusted\_price - total\_depreciation;**

**177 d │ │ call print\_line();**

**178 d │ └── end;**

**179 c │ call summary();**

**180 c └── end sum\_of\_years;**

**181 b**

**182 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**183 b /\* Эта процедура вычисляет амортизацию \*/**

**184 b /\* double\_declining. \*/**

**185 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**186 b ┌── double\_declining:**

**187 c │ procedure;**

**188 c │ adjusted\_price = selling\_price - bonus\_dep;**

**189 c │ put file (output) edit('D O U B L E D E C L I N I N G')**

**190 c │ (page,column(indent),x(10),a);**

**191 c │ call header();**

**192 c │ depreciation\_value = adjusted\_price - residual\_value;**

**193 c │ book\_value = adjusted\_price;**

**194 c │ total\_depreciation = 0;**

Листинг 16-10 Программа DEPREC (продолжение)

**195 d │ ┌── do current\_year = 1 to years**

**196 d │ │ while (depreciation\_value > 0);**

**197 d │ │ year\_value = decimal(book\_value/years,8,2) \* factor+.005;**

**198 d │ │ if current\_year = 1 then**

**199 e │ │ ┌── do;**

**200 e │ │ │ year\_value = year\_value \* months\_remaining / 12;**

**201 e │ │ │ FYD = year\_value;**

**202 e │ │ └── end;**

**203 d │ │ if year\_value > depreciation\_value then**

**204 d │ │ year\_value = depreciation\_value;**

**205 d │ │ depreciation\_value = depreciation\_value - year\_value;**

**206 d │ │ total\_depreciation = total\_depreciation + year\_value;**

**207 d │ │ book\_value = adjusted\_price - total\_depreciation;**

**208 d │ │ call print\_line();**

**209 d │ └── end;**

**210 c │ call summary();**

**211 c └── end double\_declining;**

**212 b**

**213 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**214 b /\* Эта процедура печатает вывода заголовка записи \*/**

**215 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**216 b ┌── header:**

**217 c │ procedure;**

**218 c │ declare**

**219 c │ new\_or\_used character(5);**

**220 c │**

**221 c │ ┌── if new = 'yes' then**

**222 c │ │ new\_or\_used = ' New';**

**223 c │ └── else**

**224 c │ new\_or\_used = ' Used';**

**225 c │ put file (output) edit(**

**226 c │ '--------------------------------------------------',**

**227 c │ '|',selling\_price+sales\_tax,new\_or\_used,**

**228 c │ residual\_value,' Residual Value|',**

**229 c │ '|',months\_remaining,' Months Left ',**

**230 c │ tax\_rate,'% Tax',tax\_bracket,'% Tax Bracket|')**

**231 c │ (2(skip,column(indent),a),**

**232 c │ 2(p'B$$,$$$,$$9.V99',a),**

**233 c │ skip,column(indent),a,x(5),f(2),a,2(x(2),p'B99',a));**

**234 c │**

**235 c │ put file (output) edit(**

**236 c │ '--------------------------------------------------',**

**237 c │ '| Y | Depreciation | Depreciation | Book Value |',**

**238 c │ '| r | For Year | Remaining | |',**

**239 c │ '--------------------------------------------------')**

**240 c │ (skip,column(indent),a);**

**241 c └── end header;**

Листинг 16-10 Программа DEPREC (продолжение)

**242 b**

**243 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**244 b /\* Эта процедура распечатывает текущую строку. \*/**

**245 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**246 b ┌── print\_line:**

**247 c │ procedure;**

**248 c │ put file (output) edit(**

**249 c │ '|',current\_year,**

**250 c │ ' |',year\_value,**

**251 c │ ' |',depreciation\_value,**

**252 c │ ' |',book\_value,' |')**

**253 c │ (skip,column(indent),a,f(2),4(a,p'$z,zzz,zz9v.99'));**

**254 c └── end print\_line;**

**255 b**

**256 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**257 b /\* Эта процедура распечатывает сводку значений \*/**

**258 b /\* для каждого типа графика амортизации. \*/**

**259 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**260 b ┌── summary:**

**261 c │ procedure;**

**262 c │ declare**

**263 c │ adj\_ITC decimal(8,2),**

**264 c │ total decimal(8,2),**

**265 c │ direct decimal(8,2);**

**266 c │**

**267 c │ call line();**

**268 c │ adj\_ITC = ITC \* 100 / tax\_bracket;**

**269 c │ total = FYD + sales\_tax + adj\_ITC + bonus\_dep;**

**270 c │ direct = total \* tax\_bracket / 100;**

**271 c │ put file (output) edit(**

**272 c │ '| First Year Reduction in Taxable Income |',**

**273 c │ '--------------------------------------------------',**

**274 c │ '| Depreciation ' ,FYD, '|',**

**275 c │ '| Sales Tax ' ,sales\_tax, '|',**

**276 c │ '| ITC (Adjusted) ' ,adj\_ITC, '|',**

**277 c │ '| Bonus Depreciation ' ,bonus\_dep, '|',**

**278 c │ '| ------------- |',**

**279 c │ '| Total for First Year ' ,total, '|',**

**280 c │ '| Direct Reduction in Tax ' ,direct, '|')**

**281 c │ (2(skip,column(indent),a),2(4(skip,column(indent),a,**

**282 c │ p'$z,zzz,zz9v.99',x(3),a),skip,column(indent),a));**

**283 c │ call line();**

**284 c └── end summary;**

Листинг 16-10 Программа DEPREC (продолжение)

**285 b**

**286 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**287 b /\* Эта процедура распечатывает строку тире. \*/**

**288 b /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**289 b ┌── line:**

**290 c │ procedure;**

**291 c │ put file (output) edit(**

**292 c │ '--------------------------------------------------')**

**293 c │ (skip,column(indent),a);**

**294 c └── end line;**

**295 b**

**296 b**

**297 b end depreciate;**

Листинг 16-10 Программа DEPREC (продолжение)

### Выбор графика

Есть две конструкции в DEPREC, которые заслуживают специального рассмотрения.

DEPREC использует массив переменных ENTRY, чтобы выбрать один из трех графиков. Строка 42 определяет массив с индексом в диапазоне от нуля до трех. Строки 46-49 инициализируют отдельные элементы массива и позволяют непрямые вызовы подпрограммы ERROR или подпрограммы обрабатывающей один из графиков амортизации. Позже в программе происходят фактические вызовы подпрограммам.

Выбор графика происходит на строке 74, где DEPREC читает один из символов s, y или d с консоли в символьную переменную select\_sched. Затем в строка 93 вызывается подпрограмма DISPLAY, которая выполняет вызов фактического обработчика графика с помощью оператора в строке 108:

**call schedule (index (schedules,select\_sched));**

Этот конкретный оператор работает следующим образом. Строка 43 определяет переменную schedules и инициализирует ее символьной строкой 'syd', где каждая буква соответствует одной из следующих подпрограмм обработки графика:

**syd**

**123**

**│││**

**││└────── double\_declining**

**│└─────── sum\_of\_years**

**└──────── straight\_line**

Поэтому оператор

**call schedule (index(schedules,select\_sched));**

эквивалентен:

**call schedule (index(syd,select\_sched));**

и для допустимых вводов s, y или d производит 1, 2 или 3, соответственно.

Таким образом, если select-sched равен s, оператор вызова становится:

**call schedule(1);**

и вызывает подпрограмму straight\_line. Аналогично, ввод y или d приводит к:

**call schedule(2);** или **call schedule(3);**

создавая вызовы sum\_of\_years или double\_declining, соответственно.

Если значение select\_sched не является s, y или d, то функция INDEX возвращает нулевое значение. Ввод любых недопустимых значений символов производит:

**call schedule(0);**

который вызывает подпрограмму ERROR и распечатывает сообщение об ошибке.

### Отображение выходных данных

Другая конструкция DEPREC - переменная выходного файла, определенная в строке 39. Во время фазы ввода DEPREC запрашивает вас:

**List? (yes/no)**

Ответ yes отправляет вывод программы на консоль и устройство печати.

В строке 40 объявляются, две файловые константы: sysprint и list, адресующие консоль и устройство печати. DEPREC сначала открывает консольный файл в строке 51, используя бесконечную длину страницы, чтобы избежать символов прогона страницы.

На любой итерации основной DO-группы, если вы даете утвердительный ответ в строке 77, DEPREC впоследствии открывает устройство печати, строка 78. Этот оператор может выполняться несколько раз во время конкретного выполнения программы, но только первый оператор OPEN не оказывает никакого влияния. PL/I игнорирует оператор OPEN, если файл уже открыт.

В строке 91 происходит вызов подпрограммы display, чтобы вычислить и отобразить выходной отчет для определенного набора входных значений. Подпрограмма display имеет единственный фактический параметр, состоящий из файловой константы sysprint, которая определена как формальный параметр f в строке 104. Строка 107 присваивает формальный параметр глобальной переменной output. Последующие операторы PUT пишут данные на консоль, создавая первый отчет.

В строке 92, если переменная copy\_to\_list имеет символьное значение yes, то DEPREC вызывает display еще раз. На этот раз фактический параметр - list, соответствующий системному устройству печати. Таким образом, переменной выходного файла косвенно присваивается значение list и все операторы PUT, относящиеся к файлу output, отправляют данные на принтер. Это создает электронную и бумажную копии отчета.

DEPREC использует несколько различных форм десятичной системы исчисления. Исследуйте различные объявления при перекрестной проверке выходных форматов с отображаемыми результатами.

A>**deprec**

Depreciation Schedule

Selling Price? **200000**

Residual Value? **40000**

Sales Tax (%)? **6**

Tax Bracket(%)? **50**

ProRate Months? **10**

How Many Years? **7**

New? (yes/no) **no**

Schedule:

Straight (s)

Sum-of-Yrs (y)

Double Dec (d)? **d**

List? (yes/no) **no**

D O U B L E D E C L I N I N G

--------------------------------------------------

| $212,000.00 Used $40,000.00 Residual Value|

| 10 Months Left 06% Tax 50% Tax Bracket|

--------------------------------------------------

| Y | Depreciation | Depreciation | Book Value |

| r | For Year | Remaining | |

--------------------------------------------------

| 1 |$ 35,357.14 |$ 122,642.86 |$ 162,642.86 |

| 2 |$ 34,852.04 |$ 87,790.82 |$ 127,790.82 |

| 3 |$ 27,383.75 |$ 60,407.07 |$ 100,407.07 |

| 4 |$ 21,515.79 |$ 38,891.28 |$ 78,891.28 |

| 5 |$ 16,905.27 |$ 21,986.01 |$ 61,986.01 |

| 6 |$ 13,282.71 |$ 8,703.30 |$ 48,703.30 |

| 7 |$ 8,703.30 |$ 0.00 |$ 40,000.00 |

--------------------------------------------------

| First Year Reduction in Taxable Income |

--------------------------------------------------

| Depreciation $ 35,357.14 |

| Sales Tax $ 12,000.00 |

| ITC (Adjusted) $ 20,000.00 |

| Bonus Depreciation $ 2,000.00 |

| ------------- |

| Total for First Year $ 69,357.14 |

| Direct Reduction in Tax $ 34,678.57 |

--------------------------------------------------

Листинг 16‑11 Первое взаимодействие с DEPREC

Depreciation Schedule

Selling Price? **,**

Residual Value? **,**

Sales Tax (%)? **,**

Tax Bracket(%)? **,**

ProRate Months? **8**

How Many Years? ,

New? (yes/no) **yes**

Schedule:

Straight (s)

Sum-of-Yrs (y)

Double Dec (d)? **y**

List? (yes/no) **no**

S U M O F T H E Y E A R S

--------------------------------------------------

| $212,000.00 New $40,000.00 Residual Value|

| 8 Months Left 06% Tax 50% Tax Bracket|

--------------------------------------------------

| Y | Depreciation | Depreciation | Book Value |

| r | For Year | Remaining | |

--------------------------------------------------

| 1 |$ 26,333.33 |$ 131,666.67 |$ 171,666.67 |

| 2 |$ 28,214.29 |$ 103,452.38 |$ 143,452.38 |

| 3 |$ 18,473.64 |$ 84,978.74 |$ 124,978.74 |

| 4 |$ 12,139.82 |$ 72,838.92 |$ 112,838.92 |

| 5 |$ 7,804.17 |$ 65,034.75 |$ 105,034.75 |

| 6 |$ 4,645.34 |$ 60,389.41 |$ 100,389.41 |

| 7 |$ 2,156.76 |$ 58,232.65 |$ 98,232.65 |

--------------------------------------------------

| First Year Reduction in Taxable Income |

--------------------------------------------------

| Depreciation $ 26,333.33 |

| Sales Tax $ 12,000.00 |

| ITC (Adjusted) $ 40,000.00 |

| Bonus Depreciation $ 2,000.00 |

| ------------- |

| Total for First Year $ 80,333.33 |

| Direct Reduction in Tax $ 40,166.66 |

--------------------------------------------------

Листинг 16‑12 Второе взаимодействие с DEPREC

Depreciation Schedule

Selling Price? **310000**

Residual Value? **30000**

Sales Tax (%)? **,**

Tax Bracket(%)? **,**

ProRate Months? **12**

How Many Years? **5**

New? (yes/no) **yes**

Schedule:

Straight (s)

Sum-of-Yrs (y)

Double Dec (d)? **d**

List? (yes/no) **no**

D O U B L E D E C L I N I N G

--------------------------------------------------

| $328,600.00 New $30,000.00 Residual Value|

| 12 Months Left 06% Tax 50% Tax Bracket|

--------------------------------------------------

| Y | Depreciation | Depreciation | Book Value |

| r | For Year | Remaining | |

--------------------------------------------------

| 1 |$ 123,200.00 |$ 154,800.00 |$ 184,800.00 |

| 2 |$ 73,920.00 |$ 80,880.00 |$ 110,880.00 |

| 3 |$ 44,352.00 |$ 36,528.00 |$ 66,528.00 |

| 4 |$ 26,611.20 |$ 9,916.80 |$ 39,916.80 |

| 5 |$ 9,916.80 |$ 0.00 |$ 30,000.00 |

--------------------------------------------------

| First Year Reduction in Taxable Income |

--------------------------------------------------

| Depreciation $ 123,200.00 |

| Sales Tax $ 18,600.00 |

| ITC (Adjusted) $ 62,000.00 |

| Bonus Depreciation $ 2,000.00 |

| ------------- |

| Total for First Year $ 205,800.00 |

| Direct Reduction in Tax $ 102,900.00 |

--------------------------------------------------

Листинг 16‑13 Третье взаимодействие с DEPREC

Depreciation Schedule

Selling Price? **,**

Residual Value? **,**

Sales Tax (%)? **,**

Tax Bracket(%)? **,**

ProRate Months? **,**

How Many Years? **,**

New? (yes/no) **,**

Schedule:

Straight (s)

Sum-of-Yrs (y)

Double Dec (d)? **s**

List? (yes/no) **,**

S T R A I G H T L I N E

--------------------------------------------------

| $328,600.00 New $30,000.00 Residual Value|

| 12 Months Left 06% Tax 50% Tax Bracket|

--------------------------------------------------

| Y | Depreciation | Depreciation | Book Value |

| r | For Year | Remaining | |

--------------------------------------------------

| 1 |$ 55,600.00 |$ 222,400.00 |$ 252,400.00 |

| 2 |$ 44,480.00 |$ 177,920.00 |$ 207,920.00 |

| 3 |$ 35,584.00 |$ 142,336.00 |$ 172,336.00 |

| 4 |$ 28,467.20 |$ 113,868.80 |$ 143,868.80 |

| 5 |$ 22,773.76 |$ 91,095.04 |$ 121,095.04 |

--------------------------------------------------

| First Year Reduction in Taxable Income |

--------------------------------------------------

| Depreciation $ 55,600.00 |

| Sales Tax $ 18,600.00 |

| ITC (Adjusted) $ 62,000.00 |

| Bonus Depreciation $ 2,000.00 |

| ------------- |

| Total for First Year $ 138,200.00 |

| Direct Reduction in Tax $ 69,100.00 |

--------------------------------------------------

Листинг 16‑14 Четвертое взаимодействие с DEPREC

Ссылки: Справочное руководство разделы 3.1, 3.5, 4.2, 11.3

**Конец раздела 16**

# Подпрограммы динамического распределение памяти и стека

В этом разделе описываются некоторые функции библиотеки подпрограмм времени выполнения PL/I, которые выполняют динамическое распределение памяти и управляют размером стека.

## Подпрограммы динамического распределение памяти

Библиотека подпрограмм времени выполнения включает много функций, которые обеспечивают доступ к подпрограммам динамического распределения памяти. Эти подпрограммы поддерживают связанный список всей нераспределенной памяти. По запросу эти подпрограммы ищут первый доступный сегмент в списке свободной памяти, который удовлетворяет запрашиваемому размеру, удаляют требуемый сегмент и возвращают оставшуюся часть свободной памяти в списке. Если область памяти не доступна, система времени выполнения устанавливает состояние ERROR(7), свободное пространство исчерпано.

PL/I динамически выделяет память при входе в рекурсивные процедуры, при обработке явных или неявных операторов OPEN для файлов, выполняющих дисковый ввод-вывод или при обработке оператора ALLOCATE. PL/I всегда выделяет четное число байтов или целых слов, вне зависимости от запрошенного размера.

### Функции TOTWDS и MAXWDS

Часто бывает полезно, найти объем памяти, доступный в любой заданной точке во время работы программы. Функции TOTWDS (Total Words) и MAXWDS (Max Words) предоставляют эту информацию.

Необходимо объявить функции в программе вызова следующим образом:

**declare totwds entry returns(fixed(15));**

**declare maxwds entry returns(fixed(15));**

Когда вы вызываете подпрограмму TOTWDS, она сканирует список свободной памяти и возвращает общее количество доступных слов (двойных байтов). Подпрограмма MAXWDS возвращает размер (в словах) самого большого непрерывного сегмента в списке свободной памяти. Последующий оператор ALLOCATE, определяет размер сегмента, меньше или равный MAXWDS, не сигнализирует ERROR(7), потому что, по крайней мере, такой объем памяти доступен.

Обе функции TOTWDS и MAXWDS оперируют словами, поэтому возвращаемые значения могут быть сохранены счетчиками FIXED BINARY(15). Обе функции TOTWDS и MAXWDS возвращают значение -1, если они сталкиваются с недопустимыми ссылками слов во время сканирования списка свободной памяти. Обычно, этот возврат происходит при выходе за допустимый диапазон индекса или указателя при выполнении операции. В противном случае, эти функции возвращают неотрицательное целочисленное значение.

### Подпрограмма ALLWDS

Библиотека подпрограмм времени выполнения PL/I содержит подпрограмму, названную ALLWDS, которая используется для управления размером динамического выделения. Подпрограмму необходимо объявить в вызывающей программе следующим образом:

**declare allwds entry(fixed(15)) returns(pointer);**

Подпрограмма ALLWDS выделяет сегмент памяти в словах, равных размеру, данному входным параметром, и возвращает указатель на выделенный сегмент. Если сегмент не доступен, ALLWDS устанавливает состояние ERROR(7). Входное значение должно быть неотрицательным целым числом.

*Листинг 17-1* показывает программу ALLTST, которая является примером использования функций TOTWDS, MAXWDS и ALLWDS. *Листинг 17-2* показывает пример взаимодействия с программой ALLTST.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\*** Эта программа тестирует функции TOTWDS, MAXWDS и  **\*/**

**3 a /\*** ALLWDS из библиотеки подпрограмм времени выполнения**.\*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── alltst:**

**6 b │ procedure options(main);**

**7 b │ declare**

**8 b │ totwds returns(fixed(15)),**

**9 b │ maxwds returns(fixed(15)),**

**10 b │ allwds entry(fixed(15)) returns(pointer);**

**11 b │**

**12 b │ declare**

**13 b │ allreq fixed(15),**

**14 b │ memptr ptr,**

**15 b │ meminx fixed(15),**

**16 b │ memory (0:0) bit(16) based(memptr);**

**17 b │**

**18 c │ ┌── do while('1'b);**

**19 c │ │ put edit (totwds(),' Total Words Available',**

**20 c │ │ maxwds(),' Maximum Segment Size',**

**21 c │ │ 'Allocation Size? ') (2(skip,f(6),a),skip,a);**

**22 c │ │ get list(allreq);**

**23 c │ │ memptr = allwds(allreq);**

**24 c │ │ put edit('Allocated',allreq,' Words at ',unspec(memptr))**

**25 c │ │ (skip,a,f(6),a,b4);**

**26 c │ │**

**27 c │ │ /\* очистить память как пример \*/**

**28 d │ │ ┌── do meminx = 0 to allreq-1;**

**29 d │ │ │ memory(meminx) = '0000'b4;**

**30 d │ │ └── end;**

**31 c │ └── end;**

**32 b │**

**33 b └── end alltst;**

Листинг 17‑1 Программа ALLTST

A>**alltst**

21026 Total Words Available

21026 Maximum Segment Size

Allocation Size? **0**

Allocated 0 Words at 2BBE

21024 Total Words Available

21024 Maximum Segment Size

Allocation Size? **100**

Allocated 100 Words at 2BC2

20922 Total Words Available

20922 Maximum Segment Size

Allocation Size? **500**

Allocated 500 Words at 2C8E

20420 Total Words Available

20420 Maximum Segment Size

Allocation Size? **20421**

ERROR (7), Free Space Exhausted

Traceback: 016D

A>

Листинг 17‑2 Взаимодействие с программой ALLTST

## Функция STKSIZ

В PL/I стек программы помещен выше области кода и данных, и ниже области динамической памяти (TPA). Размер стека программы по умолчанию составляет 512 байтов, но может быть изменен, используя опцию STACK(n) в заголовке главной процедуры.

Функция STKSIZ (Stack Size) возвращает текущий размер стека в байтах. Эта функция особенно полезна для проверки возможных условий переполнения стека, или для определения максимальной глубины стека во время тестирования программы.

Подпрограмму STKSIZ необходимо объявить в вызывающей программе следующим образом:

**declare stksiz returns(fixed(15));**

*Листинг 17-3* показывает пример функции STKSIZ в программе под названием ACKTST, которая проверяет максимальную глубину стека во время обработки рекурсивной процедуры. *Листинг 17-4* показывает взаимодействие с этой программой.

**1 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**2 a /\* Эта программа тестирует функцию STKSIZ при \*/**

**3 a /\* вычислении рекурсивной процедуры. \*/**

**4 a /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**5 a ┌── ack:**

**6 b │ procedure options(main,stack(2000));**

**7 b │ declare**

**8 b │ (m,n) fixed,**

**9 b │ (maxm,maxn) fixed,**

**10 b │ ncalls decimal(6),**

**11 b │ (curstack, stacksize) fixed,**

**12 b │ stksiz entry returns(fixed);**

**13 b │**

**14 b │ put skip list('Type max m,n: ');**

**15 b │ get list(maxm,maxn);**

**16 c │ ┌── do m = 0 to maxm;**

**17 d │ │ ┌── do n = 0 to maxn;**

**18 d │ │ │ ncalls = 0;**

**19 d │ │ │ curstack = 0;**

**20 d │ │ │ stacksize = 0;**

**21 d │ │ │ put edit('Ack(',m,',',n,')=',ackermann(m,n),**

**22 d │ │ │ ncalls,' Calls,',stacksize,' Stack Bytes')**

**23 d │ │ │ (skip,a,2(f(2),a),f(6),f(7),a,f(4),a);**

**24 d │ │ └── end;**

**25 c │ └── end;**

**26 b │ stop;**

**27 b │**

**28 b │ ┌── ackermann:**

**29 c │ │ procedure(m,n) returns(fixed) recursive;**

**30 c │ │**

**31 c │ │ declare**

**32 c │ │ (m,n) fixed;**

**33 c │ │ ncalls = ncalls + 1;**

**34 c │ │ curstack = stksiz();**

**35 c │ │ if curstack > stacksize then**

**36 c │ │ stacksize = curstack;**

**37 c │ │ if m = 0 then**

**38 c │ │ return(n+1);**

**39 c │ │ if n = 0 then**

**40 c │ │ return(ackermann(m-1,1));**

**41 c │ │ return(ackermann(m-1,ackermann(m,n-1)));**

**42 c │ └── end ackermann;**

**43 b │**

**44 b └── end ack;**

Листинг 17‑3 Программа ACKTST

A>**acktst**

Type max m,n: **6,6**

Ack( 0, 0)= 1 1 Calls, 4 Stack Bytes

Ack( 0, 1)= 2 1 Calls, 4 Stack Bytes

Ack( 0, 2)= 3 1 Calls, 4 Stack Bytes

Ack( 0, 3)= 4 1 Calls, 4 Stack Bytes

Ack( 0, 4)= 5 1 Calls, 4 Stack Bytes

Ack( 0, 5)= 6 1 Calls, 4 Stack Bytes

Ack( 0, 6)= 7 1 Calls, 4 Stack Bytes

Ack( 1, 0)= 2 2 Calls, 6 Stack Bytes

Ack( 1, 1)= 3 4 Calls, 8 Stack Bytes

Ack( 1, 2)= 4 6 Calls, 10 Stack Bytes

Ack( 1, 3)= 5 8 Calls, 12 Stack Bytes

Ack( 1, 4)= 6 10 Calls, 14 Stack Bytes

Ack( 1, 5)= 7 12 Calls, 16 Stack Bytes

Ack( 1, 6)= 8 14 Calls, 18 Stack Bytes

Ack( 2, 0)= 3 5 Calls, 10 Stack Bytes

Ack( 2, 1)= 5 14 Calls, 14 Stack Bytes

Ack( 2, 2)= 7 27 Calls, 18 Stack Bytes

Ack( 2, 3)= 9 44 Calls, 22 Stack Bytes

Ack( 2, 4)= 11 65 Calls, 26 Stack Bytes

Ack( 2, 5)= 13 90 Calls, 30 Stack Bytes

Ack( 2, 6)= 15 119 Calls, 34 Stack Bytes

Ack( 3, 0)= 5 15 Calls, 16 Stack Bytes

Ack( 3, 1)= 13 106 Calls, 32 Stack Bytes

Ack( 3, 2)= 29 541 Calls, 64 Stack Bytes

Ack( 3, 3)= 61 2432 Calls, 128 Stack Bytes

Ack( 3, 4)= 125 10307 Calls, 256 Stack Bytes

Ack( 3, 5)= 253 42438 Calls, 512 Stack Bytes

Ack( 3, 6)=

...

Листинг 17‑4 Вывод программы ACKTST

**Конец раздела 17**

# Оверлейные программы

В этом разделе описывается использование редактора связей, для создания оверлейных программ PL/I. Оверлеи представляют собой программы, состоящий из отдельных файлов. Преимущество оверлейных программ состоит в том, что они совместно используют те же ячейки памяти, таким образом, вы можете писать большие программы, которые выполняются в среде с ограниченной памятью.

## Использование оверлейных программ PL/I

В 8-разрядных и в 16-разрядных версиях, размер области переходных программ (TPA) определяет верхний предел размера программы. Однако 16-разрядные версии имеют другое ограничение. Несмотря на то, что объем памяти, доступной в системе, может быть достаточно, компилятор генерирует код, который предполагает малую модель памяти. Малая модель означает, что, когда вы объединяете один или несколько объектных файлов с библиотекой времени выполнения, размер каждой из секций кода и данных в CMD или EXE ограничен 64K. Таким образом, компилятор определяет верхний предел размера любой программы, но не сталкивается с ограничением размера до момента компоновки.

Благодаря модульной конструкции, вы можете писать большие программы, которые не должны находиться в памяти одновременно. Например, много прикладных программ управляются с помощью меню, в котором пользователь выбирает для выполнения одну из нескольких функций. Поскольку функции отдельные и вызываются последовательно, они не должны находиться в памяти одновременно. Когда одна из функций завершается, управление возвращается к части программы с меню, из которой пользователь выбирает следующую функцию. Используя оверлейные программы, вы можете разделить такую программу на отдельные подпрограммы, которые могут быть сохранены на диске и загружены только при необходимости.

Следующий рисунок иллюстрирует концепцию оверлейных программ.



Рисунок 18‑1 Использование оверлея в большой программе

Предположим, что управляемая с помощью меню прикладная программа состоит из трех отдельных, выбираемых пользователем функций. Если каждая функция требует 30K памяти, и часть меню требует 10K, то общая память, требуемая для программы, составляет 100K, как показано на *Рисунке 18-1а*. Однако, если три функции разработаны как оверлейные программы, как показано на *Рисунке 18-1б*, программа требует только 40K, потому что все три функции совместно используют одни и те же ячейки памяти.

Вы можете также создать вложенные оверлейные программы в форме древовидной структуры, где каждое наложение может вызвать другие оверлейные программы до максимального уровня вложенности, который определяет менеджер оверлеев. *Раздел 18.3* описывает синтаксис командной строки для создания вложенных оверлейных программ.

*Рисунок 18-2* иллюстрирует древовидную структуру оверлейных программ. Верхняя часть самого высокого оверлея определяет необходимый объем памяти. *На Рисунке 18-2*, самый высокий оверлей SUB4. Это значительно меньше памяти, чем потребуется, если все функции и подфункции должны находиться в памяти одновременно.



Рисунок 18‑2 Древовидная структура оверлея

## Написание оверлейных программ PL/I

Есть два способа написания программам PL/I, использующих оверлеи. Первый способ не предполагает специального кодирования, но имеет два ограничения. Первое ограничение – все оверлеи должны находиться на диске по умолчанию. Второе - имена оверлеев должны быть определены во время трансляции и не могут быть изменены во времени выполнения.

Второй метод требует более сложную последовательность вызова, но не имеет ни одного из ограничений первого метода.

### Первый способ наложения

Чтобы использовать первый способ, оверлей следует объявить в качестве константы точки входа (entry) в модуле, в котором происходит его вызов. Как константа точки входа, оверлей может иметь параметры, объявленные в списке параметров. Сам оверлей является просто процедурой или группой процедур PL/I.

Например, следующая программа является корневым модулем с одним наложением:

**root:  
 procedure options(main);  
 declare  
 ovlay1 entry(character(15));  
 put skip list('root');  
 call ovlay1('overlay 1');  
end root;**

Наложение OVLAY1.PLI определяется следующим образом:

**ovlay1:  
 procedure(c);  
 declare  
 c character(15);  
 put skip list(c);  
end ovlay1;**

**Примечание**: При передаче параметров наложению, вы должны убедиться, что количество и тип параметров одинаковы в вызывающей программе и наложении.

При запуске программы, ROOT сначала выведет на консоль сообщение "root". Затем оператор CALL передаст управление менеджеру оверлея. Менеджер оверлея загрузит файл OVLAY1 с диска по умолчанию и передает ему управление.

Когда наложение получает управление, оно выводит на консоль сообщение "overlay 1" . Затем OVLAY1 возвращает управление оператору в программе ROOT, расположенному непосредственно за оператором CALL. Программа подложит выполнение с этого места.

Если требуемое наложение уже находится в памяти, менеджер оверлея не загружает его снова прежде, чем передать ему управление.

Первый способ имеет следующие ограничения:

* Метка в операторе вызова является подлинным именем оверлейного файла (наложения), загружаемого менеджером оверлея, следовательно, эти два имени должны быть согласованы.
* Имя точки входа наложения не нужно согласовывать с именем, используемым в последовательности вызова, однако использование того же имени позволяет избежать путаницы.
* Менеджер оверлея загружает оверлейные программы только с диска по умолчанию на момент выполнения корневого модуля. Менеджер оверлея игнорирует любые изменения на диске по умолчанию, которые происходят после того, как корневой модуль начинает выполнение.
* Имена оверлейных программ фиксированы. Чтобы изменить имена оверлейных программ, вы должны отредактировать, перекомпилировать и повторно скомпоновать программу.
* Никакие нестандартные операторы PL/I не требуются. Таким образом, вы можете отложить решение о создавать оверлейных программ до момента компоновки.

### Второй способ наложения

В некоторых приложениях может потребоваться иметь большую гибкость с оверлейными программами, например, загрузка оверлеев из разных дисков или определение имени оверлеев из консоли или дискового файла во времени выполнения.

Чтобы сделать это, программа PL/I должна явно объявить точку входа в менеджер оверлея, следующим образом:

**declare ?ovlay entry(character(10),fixed(1));**

Эта точка входа требует два параметра. Первый являются 10 символьной строкой, которая определяет имя загружаемого оверлея и необязательный код диска в стандартном формате (d:filename).

Второй параметр - флаг загрузки. Если флаг загрузки равняется 1, менеджер оверлея загружает указанное наложение, вне зависимости от наличия его в памяти. Если флаг загрузки 0, менеджер оверлея загружает наложение, только если оно не находится в памяти.

С помощью этого метода, пример, иллюстрирующий первый метод, выглядит следующим образом:

**root:  
 procedure options(main);  
 declare  
 ?ovlay entry(character(10),fixed(1)),  
 dummy entry(character(15)),  
 name character(10);  
  
 put skip list('root');  
 name = 'OV1';  
 call ?ovlay(name,0);  
 call dummy('overlay 1');  
end root;**

Файл OV1.PLI такой же, как и в предыдущем примере.

Во времени выполнения, оператор:

**call ?ovlay(name,0);**

направляет менеджер оверлея загрузить OV1 с диска по умолчанию (1 текущее значение переменной name). Затем управление передается OV1. Когда OV1 заканчивает работу, управление возвращается оператору после вызова.

В этом примере переменной name присвоено значение 'OV1'. Однако, вы также можете указать имя оверлея в виде символьной строки из любого другого источника, например, с клавиатуры консоли.

Следующие ограничения применяются к второму способу наложения:

* Вы можете определить код диска, таким образом, менеджер оверлея может загрузить оверлейные программы с дисков отличных от диска по умолчанию. Если вы не определяете код диска, менеджер оверлея использует диск по умолчанию, как описано в первом методе.
* Если вы передаете какие-либо параметры оверлею, они должны быть согласованы по числу и типам с параметрами, которые ожидает оверлей.

### Общие ограничения наложения

Следующие общие ограничения применяются при создании оверлейных программ в программе PL/I:

* Каждое наложение имеет только одну точку входа. Менеджер оверлея в библиотеке времени выполнения подпрограммы PL/I предполагает, что эта точка входа является адресом загрузки наложения.
* Вы не можете сделать восходящую ссылку из модуля для точек входа в оверлеи выше на дереве. Единственным исключением является ссылка на главную точку входа оверлея. Вы можете сделать ссылки вниз на точки входа в оверлеях ниже на дереве или в корневой модуль.
* Общие сегменты (EXTERNALS в PL/I), которые объявлены в одном модуле, не могут быть инициализированы модулем выше в дереве. Редактор связей игнорирует любую попытку сделать это.
* Вы можете вложить оверлейные программы на глубину 5 уровней.
* Менеджер оверлея использует буфер по умолчанию, расположенный в 80H, таким образом, пользовательские программы не должны зависеть от данных, хранящихся в этом буфере. Обратите внимание, что в реализациях 8086, буфер по умолчанию в 80H относительно базы сегмента данных.

## Синтаксис командной строки

Чтобы определить оверлейные программы в командной строке редактора связей, включите каждую спецификацию наложения в круглые скобки. Вы можете создать оверлейные программы с LINK-80™ в одной из следующих форм:

**link root(ovl)**

**link root(ovl,part2,part3)**

**link root(ovl=partl,part2,part3)**

Первая форма производит файл OV1.OVL из файла OV1.REL. Вторая форма производит файл OV1.OVL из OV1.REL, PART2.REL и PART3.REL. Третья форма производит файл OV1.OVL из PART1.REL, PART2.REL и PART3.REL.

Для создание оверлейных программ с помощью LINK-86™, используются те же формы:

**link86 root(ovl)**

**link86 root(ovl,part2,part3)**

**link86 root(ovl=partl,part2,part3)**

Первая форма производит файл OV1.OVR из файла OV1.OBJ. Вторая форма производит файл OV1.OVR от OV1.OBJ, PART2.OBJ и PART3.OBJ. Третья форма производит файл OV1.OVR из PART1.OBJ, PART2.OBJ и PART3.OBJ.

В командной строке левая круглая скобка указывает начало новой спецификации наложения, и также указывает конец группы, предшествующей ему. Все файлы, которые будут включены в любую точку на дереве, должны появиться вместе без любых прошедших спецификаций наложения. Вы можете использовать пробелы, чтобы улучшить удобочитаемость, но не используете запятые, чтобы выделить спецификации наложения от корневого модуля или друг от друга.

Например следующая командная строка является неправильной:

**A>link root(ovl),moreroot**

Правильная команда выглядит следующим образом:

**A>link root,moreroot(ovl)**

Чтобы вложить оверлеи, их необходимо указать в командной строке, вложенными в скобки. Например следующая командная строка создает наложение показанное на *Рисунке 18-2*:

**A>link menu(funcl (subl)(sub 2)) (func2) (func3 (sub3)(sub4))**

**Конец раздела 18**

Предметный указатель

А

Агрегат 8

данных 13

Алгоритм 152

Аргументы 20

Атрибут

по умолчанию 23

BASED 15

DIRECT 63

ENVIRONMENT 24

EXTERNAL 125

INITIAL 31

KEYED 23, 60, 62

LINESIZE 25

PAGESIZE 25

PRINT 23, 53, 58

RECURSIVE 30, 111

STATIC 126

TITLE 24

VARYING 9

Атрибуты

допустимые 25

Б

Базированная переменная 31

спецификатор указателя 31

Безусловный переход 22

Бесконечный цикл 47

Библиотека подпрограмм

времени выполнения 39, 43, 125, 128, 177

Блок 4

BEGIN 4, 5, 72

PROCEDURE 5, 72

Блок параметров файла 25

В

Ввод-вывод

строчно-ориентированный 26

управляемый редактированием 26

управляемый списком 26

Вектор 72

Вложенность блоков 5

Внешнее устройство 12, 22, 24

Внутренний

стек 44

Встроенная функция 9

CEIL 151

DECIMAL 117, 136, 139, 151

DIVIDE 136, 141

INDEX 83, 172

LINENO, вспомогательная 30

LOCK 24

MOD 143

ONCODE, состояния 30

ONFILE, состояния 30

ONKEY, состояния 30

PAGENO, вспомогательная 30

ROUND 143

TRANSLATE 83

UNLOCK 24

VERIFY 83, 88

Вызов компилятора 40

Выравнивание операндов 136

Выражение 17, 20

арифметическое 119

вычислительное 17

невычислительное 17

скалярное 22

указателя 31

Вычисляемый GOTO 71, 72

Г

Генерация абсолютного кода 41

Д

Данные

арифметические 8

двоичные 23

константа 7

переменная 7

преобразование 17, 121

указателя 12

управления программой 10

ENTRY 126

ENTRY, точки входа 11

FIXED BINARY 8

FIXED DECIMAL 8

FLOAT BINARY 8

LABEL 10

Двоичный порядок 8

Дескриптор файла 25

Динамическое распределение памяти 177

Дополнение до 10 135, 136

Дополнение пробелами 27

Дополнительный код 135

З

Заголовок процедуры 128

Записи фиксированного размера 24

Знак числа 135

Значение

по умолчанию 9

И

Идентификатор файла 22, 23, 24

Иерархическая структура программы 3

Имя

идентификатора 7

командного файла 44

оверлейного файла 185

программы 44

процедуры 5

точки входа 185

файла 24, 44

функции 6

Исходный файл 39

Итерация 47

К

Категории состояний 28, 29, 44

Класс памяти 30

AUTOMATIC 30

BASED 31

PARAMETER 31

STATIC 30

Ключ 24

Ключевое слово 1

Командный файл 43, 128

Консольный ввод 44

Константа

битовой строки 10

метки 10, 71

символьной строки 10

файловая 12, 22

SYSIN 28

SYSPRINT 28

ENTRY, точки входа 6, 11

Л

Лексема (token) 86

Логические модули 3

Логический блок 10, 21

М

Мантисса 8

Маркер

страницы 23

строки 23

Массив 10, 13

меток 10

сечение 13

Масштабный коэффициент 8, 135

Математические функции 9

Машинно-зависимый код 41

Менеджер оверлея 185

точка входа 185

Метод рекурсивного спуска 41

Н

Набор данных 22

Непрерывное хранение 14

Неявное объявление 7, 10

О

Область транзитных программ 42

Обработка состояний 28, 71, 121

Обработка списка 91

Объектный файл 39

Оверлей

путь загрузки 185

флаг загрузки 186

Оверлейная программа 183

Оператор

%INCLUDE 33, 53, 128

%REPLACE 33, 48

присваивания 17, 71, 111

пустой 22

ALLOCATE 32, 91, 95, 177

CALL 5, 20, 22

CLOSE 26

DECLARE 7

DO 18

END 5

FORMAT 28

FREE 32, 96

GET 80

GET EDIT 26, 81, 88

GET LIST 26, 53, 64

GOTO 22, 72, 73

правила использования 22

IF 22, 33

NULL, пустой 33

ON 28, 75, 80

OPEN 23, 25, 28, 51, 53, 172, 177

PUT EDIT 26

PUT LIST 26, 58

READ 26, 51, 62

с KEY 63

REVERT 29, 75, 81

SIGNAL 29, 77

STOP 26, 47, 81, 121

WRITE 26, 51, 60

с KEYFROM 64

Операторы

ввода-вывода 23, 28

выполняемые 3

объявления 3, 7

препроцессора 33

структурные 3

управления 18

Операционная система 28

Опции компилятора 41

Опция

ENVIRONMENT 60

KEYTO 62

MAIN 117

STACK 117

TITLE 52

Отладка программы 35

Отступы 35

Ошибка

усечения 8, 49, 134, 140

TRUNC (усечение) 35

П

Параметры 20

Передача параметров

значением 20

ссылкой 20

Переменная

базированная 7, 17, 31, 91

битовой строки 10

метки 10, 71

строковая 9

типа POINTER 31

указателя 12, 91

файловая 12, 22

CHARACTER 9

ENTRY, точки входа 11

Поддержка программы 37

Подкод ошибки 44

Подмножество G 1

Подразумеваемая основа 91

Подразумеваемые атрибуты 25

Предопределенные файловые константы 28

Прерывание компиляции 42

Программа

отладка 5, 43

DEMO 43

Процедура

внешняя 3, 6

внутренняя 6, 11

вызов 5, 20

главная 3

заголовок 5, 20, 22

имя 5

определение 20

подпрограмма 5, 20

рекурсивная 20, 111

тело 5

функция 5, 20

Псевдопеременная 17

Р

Редактор связей 128, 183

Режим открытия файла

совместное использование 24

только для чтения 24

С

Свободная область памяти 32

Свободный формат 35

Символ

конца файла 44, 52

Скалярная величина 6

Содержащий блок 4, 22

Состояние

переполнения 136

потери значимости 136

ENDPAGE 81

FIXEDOVERFLOW 112, 137, 139

Специальные символы 2

Спецификатор указателя 91

Спецификация диска 40

Список формата 26

Среда 21, 29

Ссылка

определенная с помощью указателя 31

Ссылка на метку

локальная 72

Стек 136, 139

времени выполнения 117

Структура 14

основная 15

член 15

Структурированный язык 1

Т

Таблица символов 39, 41, 95, 125

Текстовый редактор 23

Тип файла

PLI 40

PRN 41

Типы данных PL/I 7

Точность 135

У

Указатель 31

Управляющий символ

^J 66

^M 66

Ctrl-Break 42

Ctrl-C 2, 84

Ctrl-G 81

Ctrl-I 35

Ctrl-Z 77, 80, 84

Уровень вложенности 42

Условный переход 22

Ф

Файл 22

защита паролем 24

режим открытия 24

уровень защиты

Delete 24

Read 24

Write 24

INPUT, входной 23

KEYED, с ключом 24

OUTPUT, выходной 23

RECORD, записеориентированный 23

SEQUENTIAL, последовательный 24

STREAM, потоковый 23

SYSIN 77, 80

SYSPRINT 47, 58

UPDATE, обновляемый 24

Фактические параметры 76

Фактический параметр 172

Формальные параметры 111

Формальный параметр 172

Формат битовой строки

B1 10

B2 10

B3 10

B4 10

Функциональная ссылка 6

Х

Хвост команды 52

Хранение

совместно используемое 20

Ч

Число

*двойной точности* 8

одинарной точности 8

точность 8

целое 8

Э

Элемент формата данных

A 27

B 27

E 27

F 27

Элемент формата управления

**COLUMN** 27

**LINE** 27

**PAGE** 27

**R** 28

**SKIP** 27

**X** 27

Элементы формата

данных 26

управления 26

Я

Явное объявление 7

B

BASIC 122, 133

C

COBOL 133

D

DO-группа 6, 47, 88, 95

F

FIXED BINARY 7, 23

FIXED DECIMAL 7

FORTRAN 133

O

ON блок 81, 84

ON состояние 77

ON-блок 45

ON-тело 28, 77

P

PASCAL 133

1. Оригинал руководства на английском языке [↑](#footnote-ref-1)
2. BCD сокращение от английского Binary Coded Decimal [↑](#footnote-ref-2)
3. FPB сокращение от английского File Parameter Block [↑](#footnote-ref-3)
4. TPA сокращение от английского Transient Program Area [↑](#footnote-ref-4)
5. суперкрутонавороченный [↑](#footnote-ref-5)
6. GNT сокращение от английского Get Next Token. [↑](#footnote-ref-6)